Titre: Title:	Méthodes de diagnostic d'erreurs d'analyse syntaxique
Auteur: Author:	Matthieu Ouellette-Vachon
Date:	2013
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ouellette-Vachon, M. (2013). Méthodes de diagnostic d'erreurs d'analyse syntaxique [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1130/

UNIVERSITÉ DE MONTRÉAL

MÉTHODES DE DIAGNOSTIC D'ERREURS D'ANALYSE SYNTAXIQUE

MATTHIEU OUELLETTE-VACHON
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODES DE DIAGNOSTIC D'ERREURS D'ANALYSE SYNTAXIQUE

présenté par : OUELLETTE-VACHON Matthieu
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GALINIER Philippe, Doct., président
M. GAGNON Michel, Ph.D., membre et directeur de recherche
M. BOUCHER Dominique, Ph.D., membre et codirecteur de recherche
M. MERLO Ettore, Ph.D., membre

Je m'oppose à faire les choses que les ordinateurs peuvent faire.
I object to doing things that computers can do.
— Olin Shivers

REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur de recherche, Michel Gagnon, qui m'a fortement encouragé à explorer le monde de la recherche scientifique. Merci pour la patience, l'écoute ainsi que l'aide précieuse. Tout ceci m'a permis de me rendre au bout de cette aventure.

Il me faut également souligner l'apport incroyable de mon codirecteur de recherche, Dominique Boucher. Il a été une source d'inspiration constante pour moi. Je le remercie pour l'appui inconditionnel qu'il m'a donné tout au long de ma recherche.

Merci à la compagnie Nu Echo qui m'a soutenu de plusieurs façons, notamment en fournissant les grammaires et les données d'expérimentation. De plus, je me dois de remercier tous mes collègues de travail chez Nu Echo qui ont su égayer mes journées les plus difficiles.

Également, j'aimerais dédier un paragraphe à mes parents, Johanne et François, qui ont su être présents tout au long de mes études et qui ont tout fait pour que je me rende au bout de mes projets. De plus, je me dois de remercier ma copine, Emilie. Elle a sacrifié son lot de soirée romantique afin de me permettre de terminer mon mémoire.

Finalement, j'aimerais dire un grand merci aux trois partenaires qui m'ont accordé une bourse d'étude BMP dans le cadre de cette recherche soit : le FQRNT, le CRSNG et Nu Echo.

RÉSUMÉ

Dans les applications de reconnaissance vocale, l'ensemble des phrases qu'une grammaire reconnaît a un impact important sur l'expérience utilisateur. Pour une interaction donnée, le but est d'avoir une grammaire couvrant un grand nombre de cas de figure sans être trop générale au point d'inclure des formes qui devraient être rejetées. Dans cette optique, le développement d'outils automatiques pouvant améliorer une grammaire est d'une grande utilité. Mais pour y parvenir, il faut d'abord diagnostiquer les raisons pour lesquelles une phrase n'a pu être reconnue.

Avec cet objectif en tête, nous nous sommes mis en quête d'algorithmes d'analyse syntaxique robuste. Actuellement, on peut distinguer trois grandes familles d'algorithmes, selon le contexte utilisé pour la correction : local, régional ou global.

De par leur nature, les algorithmes de correction globale offrent de meilleurs diagnostics, car ils analysent toutes les hypothèses d'erreur possible. Cependant, ceci a un coût en termes de temps de calcul. Dans le cadre de ce projet, nous avons cherché à savoir si les algorithmes offrant une correction locale peuvent rivaliser avec les algorithmes qui effectuent une correction régionale ou globale au niveau de la qualité des corrections.

Nous avons évalué trois algorithmes : l'algorithme de Lyon, qui effectue une correction globale, l'algorithme d'Anderson-Backhouse, qui effectue une correction locale, et finalement un algorithme que nous avons développé, qui pourrait être classé parmi la famille des algorithmes de correction régionale. Ce dernier est un hybride entre l'algorithme de Lyon et l'algorithme d'Earley.

À l'aide d'une méthodologie bien précise, nous avons confirmé que l'algorithme de correction globale effectue les meilleures corrections, mais avec une différence moyenne de seulement 3.96 %. L'algorithme de correction locale est cependant environ 2 et 4 fois plus rapide respectivement que l'algorithme hybride et l'algorithme de correction globale.

À la lumière de ces résultats, nous avons conclu que l'algorithme effectuant une correction locale peut avantageusement se comparer à l'algorithme de correction globale, mais en étant nettement plus rapide. Cette conclusion s'inscrit dans le contexte de notre l'application, elle y est donc peut-être limitée.

Grâce à la bonne performance de l'algorithme de correction locale, nous envisageons de l'utiliser dans le cadre de travaux futurs. Nous prévoyons tirer profit de sa vitesse afin d'analyser un ensemble relativement grand de phrases et d'inférer les meilleures améliorations à apporter à la grammaire d'une manière totalement automatique.

ABSTRACT

In speech recognition applications, the set of sentences recognized by a grammar has a significant impact on the user experience. For a given interaction, the goal is to have a grammar covering a large number of scenarios without being general to the point of accepting forms that should be rejected. In this context, the development of automatic tools that can improve grammar coverage is utterly important. But to achieve this, we must first diagnose why a sentence hasn't been recognized.

With this goal in mind, we set out to find robust parsing algorithms. Currently, there are three main families of algorithms, depending on the context used for the correction: local, regional or global.

By their nature, the global correction algorithms provide better diagnosises because they analyze all possible error hypotheses. However, this has a cost in terms of time elapsed for an analysis. In this project, we investigated whether the local correction algorithms can compete with algorithms that perform a regional or global correction on the quality of the corrections.

We evaluated three algorithms: Lyon's algorithm, which performs a global correction of errors, Anderson-Backhouse's algorithm, which performs a local correction of errors and a hybrid algorithm. The third one is a mix between Lyon's algorithm and a classic parsing algorithm that we have developed. It could be ranked among the family of regional correction algorithms.

Using a strict methodology, we confirm that the global correction algorithm performs the best, but with a mean difference of only 3.96%. However, the local correction algorithm is respectively about 2 and 4 times faster than the hybrid algorithm and the global correction algorithm.

In light of these results, we concluded that the algorithm performing local correction of errors may advantageously be compared to the algorithm doing global correction of errors but is much faster. This conclusion is in the context of our application, it could therefore be limited to it.

Thanks to the good corrections offered by the local correction algorithm, we plan to use them in future work. We expect to use its speed to analyze a relatively large set of sentences and infer the best improvements that could be applied to the grammar in a full automatic manner.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

LISTE DES SIGLES ET ABRÉVIATIONS

ABNF	*Augmented Backus–Naur Form*
ATN	*Augmented Transition Network*
CFG	*Context-Free Grammar*
CG	*Constraint Grammar*
EOL	*End Of Line*
GLR	*Generalized Left-to-right, Rightmost derivation*
ING	*In-Grammar*
IVR	*Interactive Voice Response*
LALR	*Look-Ahead LR parser*
LL	*Left-to-right, Leftmost derivation*
LR	*Left-to-right, Rightmost derivation*
PG	*Property Grammar*
OOG	*Out-Of-Grammar*
TTS	*Text-To-Speech*
XML	*Extensible Markup Language*

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

Les applications commerciales de reconnaissance vocale utilisent principalement des dialogues dirigés pour interagir avec l'utilisateur. Ces dialogues ont la particularité de ne permettre qu'un nombre restreint de réponses possibles à chaque interaction. L'ensemble des réponses possibles est généralement décrit à l'aide d'une grammaire hors contexte, ou *Context-Free Grammar* (CFG) en anglais, qui sera utilisée par l'engin de reconnaissance vocale lors de l'exécution de l'application. Ces grammaires doivent couvrir le plus large éventail de phrases possible pour chaque dialogue, sans toutefois permettre des phrases qui ne sont pas d'usage courant, ceci afin d'obtenir des performances de reconnaissance optimales. Les outils d'aide à la mise au point de ces grammaires sont donc d'une importance primordiale.

Derrière ce dilemme d'équilibre entre une grammaire trop spécifique et une grammaire trop générale se trouve le développeur de grammaires. Cette personne est au coeur du processus itératif qu'est le développement d'une grammaire utilisable dans une application de reconnaissance vocale. En effet, la première ébauche de la grammaire est souvent approximative et il devient important dans les phases subséquentes d'évolution de bien cerner les problèmes potentiels pouvant nuire à la performance de la grammaire. Il faut à la fois augmenter la quantité de phrases bien formées qui n'ont pas été reconnues et diminuer la quantité de phrases mal formées qui sont reconnues, mais qui ne devraient pas l'être.

Au départ, le développeur de grammaires commence son travail à l'aide de la spécification du dialogue et un ensemble de phrases types devant être reconnues par la grammaire. Grâce à ces informations, il crée une version initiale de la grammaire. Après cette première version, la grammaire doit être améliorée afin d'en augmenter la couverture car il y a toujours des phrases types qui n'avaient pas été prévues au départ.

Pour y arriver, un ensemble représentatif de phrases est obtenue à partir des réponses textuelles dîtes par les utilisateurs pour une interaction qui utilise la dîtes grammaire. Le travail du développeur revient alors à trouver des manières d'améliorer la couverture de la grammaire. Des dizaines voire des centaines de phrases doivent être analysées afin de déceler certaines phrases posant problème ce qui demande une analyse très pointue.

Même en ayant en main la liste des phrases qui ne sont pas reconnues, le travail du développeur reste ardu. Avec des grammaires de plus en plus complexes, il devient difficile de trouver les parties de la grammaire qui doivent être modifiées afin d'en augmenter la couverture sans quelle devienne trop générale. Le diagnostic des erreurs de syntaxe est une

avenue intéressante pour aider le développeur de grammaire dans sa quête d'optimisation.

L'utilisation de techniques d'analyse syntaxique robuste est un point de départ essentiel pour le diagnostic des erreurs de syntaxe. Informellement, un analyseur syntaxique robuste est décrit comme un analyseur syntaxique qui est en mesure de déterminer la structure syntaxique d'une phrase qui contient une ou plusieurs erreurs et de fournir une version corrigée de celle-ci. À l'avenir, le terme analyse syntaxique robuste réfèrera à cette définition [1].

Il y a donc un lien indissociable direct entre l'analyse syntaxique robuste et le diagnostic des erreurs de syntaxe. Ce dernier ne peut être effectué sans l'apport du premier. Pour avoir de bons outils de diagnostic des erreurs, les techniques d'analyse syntaxique robuste doivent être de bonne qualité. En ce qui a trait aux outils de diagnostic d'erreurs, on distingue deux grandes classes d'outils.

La première regroupe les outils permettant de diagnostiquer les erreurs pour une seule phrase à la fois, ce qui peut prendre plusieurs formes. Par exemple, textuellement, à l'aide d'une liste des erreurs de syntaxe et leurs positions, à la manière d'un compilateur. Ou bien, visuellement, à l'aide d'un arbre partiel représentant la structure reconnue et annotée aux endroits où les erreurs se sont produites.

La deuxième classe regroupe plutôt les outils permettant de diagnostiquer les erreurs en utilisant un ensemble de phrases. Dans cette situation, plusieurs patrons d'erreurs seraient agrégés afin de faire ressortir les erreurs de syntaxe qui donneraient le meilleur rendement si elles étaient corrigées dans la grammaire.

Le développement de grammaires de reconnaissance est une tâche ardue et sujette aux erreurs. Dans cette optique, les outils d'aide à la mise au point de ces grammaires sont donc d'une grande importance afin d'obtenir des grammaires de grande qualité tout en réduisant le temps de développement.

1.1 Éléments de la problématique

Un problème récurrent pour le développeur de grammaires est de déterminer la raison pour laquelle une phrase ne donne lieu à aucune analyse grammaticale. Ce problème peut survenir lors du développement initial de la grammaire, ou bien lors des phases d'optimisation de l'application, à la lumière des phrases transcrites à partir d'interactions réelles avec l'application.

Déterminer les raisons pour lesquelles une phrase n'est pas reconnue est bien sûr l'élément principal de la problématique que nous tenterons de résoudre au cours de ce projet de recherche. Pour parvenir à proposer des outils de diagnostic performants, d'autres problèmes

1. Voir 2.2 pour une définition plus stricte

plus élémentaires doivent être abordés.

Lorsqu'une phrase n'est pas reconnue, il est primordial de déterminer où se trouvent les différentes erreurs dans la phrase. Pour la détection des erreurs, le recours à des analyseurs syntaxiques robustes nous semble un choix logique.

Ces analyseurs syntaxiques se différencient des analyseurs classiques par leur habileté à traiter des phrases contenant une ou plusieurs erreurs et à identifier les endroits où elles sont survenues. Bien sûr, il existe une panoplie d'analyseurs permettant de faire de la correction d'erreur. Le problème est alors de déterminer quels analyseurs robustes permettraient d'offrir les meilleures corrections tout en respectant les contraintes du projet.

La première contrainte est qu'il est impératif que toutes les techniques mises en place puissent fonctionner avec toutes les classes de grammaires CFG. Ceci veut donc dire que des algorithmes qui fonctionnent avec une classe particulière de grammaire hors-contexte, par exemple les grammaires *Left-to-right, Rightmost derivation* (LR)(k), ne peuvent être utilisés. La raison provient du fait que dans notre cadre applicatif, nous n'avons pas le contrôle sur la façon dont est bâtie la grammaire autre le fait qu'elle est de type CFG.

La deuxième contrainte est que les outils de diagnostic doivent être indépendants de toute intervention humaine les techniques. C'est donc dire que les techniques requérant d'ajouter des metas-informations aux grammaires sont à éviter.

Dans le monde de l'analyse syntaxique robuste, on peut distinguer trois grandes familles d'algorithmes, selon le contexte utilisé pour la correction : locale, régionale et globale.

Toutes ces techniques comportent leurs avantages et leurs désavantages, notamment pour ce qui est de la qualité de la correction et du temps requis pour analyser une phrase. La problématique est alors de déterminer les algorithmes répondant aux critères et contraintes tout en offrant une qualité de détection des erreurs acceptable. En d'autres mots, il s'agit de trouver le meilleur équilibre entre qualité de correction et performance.

L'analyse syntaxique robuste est une méthode de diagnostic des erreurs en soit. En effet, de telles techniques sont non seulement robustes aux erreurs, elles permettent également de déterminer à quels endroits dans la phrases elles sont survenues et de proposer une correction. Avec ces informations, il devient aisé de développer un outil permettant de convertir les erreurs trouvées en suggestions de correction pour la grammaire.

Cependant, appliquer ces suggestions de correction une à la suite de l'autre cause une généralisation trop importante de la grammaire, car bien des suggestions sont non pertinentes dans le cadre du dialogue couvert par la grammaire. La clé pour augmenter la qualité d'une grammaire de reconnaissance n'est donc pas simplement d'apporter des suggestions de réparation isolées. Il faut plutôt agréger l'information récoltée par l'analyseur syntaxique robuste sur un ensemble relativement important de phrases afin de faire ressortir les diagnostics

d'erreurs les plus pertinents. Le problème revient donc à être capable d'extraire des schémas récurrents d'erreur parmi un ensemble de phrases corrigées par un algorithme d'analyse syntaxique robuste.

Pour le développeur de grammaires, avoir accès à un ensemble plus petit de diagnostics contenant un maximum de suggestions pertinentes est d'une grande utilité. Ces diagnostics serviront à optimiser le taux de reconnaissance des phrases bien formées tout en minimisant la reconnaissance de phrases mal formées.

1.2 Objectif de recherche

Le but final de la recherche est de concevoir et d'évaluer des techniques de diagnostic d'erreurs d'analyse syntaxique. Ces techniques permettront à terme au développeur d'une grammaire hors contexte de comprendre les raisons pour lesquelles une phrase ne fait pas partie du langage généré par la grammaire. De plus, elles proposeront dans la mesure du possible des modifications à la grammaire afin que la phrase puisse être reconnue.

L'analyse syntaxique robuste est le premier jalon important de la problématique devant être attaqué. Une analyse syntaxique robuste affichant une faible détection des erreurs aura un effet boule de neige sur l'extraction des diagnostics d'erreurs pertinents. L'objectif de la présente recherche est donc d'évaluer différents algorithmes d'analyse syntaxique robuste afin de déterminer quelle technique serait la plus apte à être utilisée pour diagnostiquer les erreurs.

La littérature sur le sujet, notamment (Mauney et Fischer, 1988) et (Vilares *et al.*, 2000), indique clairement que les techniques d'analyse syntaxique robuste dites global offrent la meilleure qualité de correction tout en offrant la moins bonne performance en terme de temps de calcul. À l'opposé, les techniques dites de correction locale des erreurs offrent une qualité moindre de correction tout en ayant un temps d'exécution plus rapide. Le temps de calcul ayant une importance dans le domaine d'application envisagé pour ce projet de recherche, nous cherchons à savoir si une technique de correction locale, plus rapide, offre une qualité de correction semblable à une technique effectuant une correction globale des erreurs.

L'hypothèse de recherche pour cet objectif est : « les techniques de correction d'erreur locale peuvent rivaliser avec les techniques de correction d'erreur globale au niveau de la qualité des corrections ».

1.3 Plan du mémoire

La suite du mémoire se divise en cinq chapitres.

Le chapitre 2 traite de l'état de la science en ce qui a trait à l'analyse syntaxique robuste.

On y aborde les concepts de base de l'analyse syntaxique classique et une description détaillée de l'algorithme d'Earley y est présentée. Cet algorithme est à la base des algorithmes d'analyse syntaxique robuste évalués dans ce travail.

Le chapitre 3 porte sur l'analyse syntaxique robuste. Dans ce chapitre, on discute en détail des trois algorithmes d'analyse syntaxique robuste qui ont été évalués dans le cadre de ce projet : l'algorithme d'Anderson-Backhouse (correction locale des erreurs), l'algorithme de Lyon (correction globale des erreurs) et finalement un algorithme hybride que nous avons développé.

Le chapitre 4 aborde la méthodologie utilisée pour l'évaluation des algorithmes. On y parle notamment des mesures utilisées pour déterminer quel algorithme offre la meilleure performance autant en terme de qualité de correction qu'en terme de temps écoulé.

Le chapitre 5 fait la synthèse des résultats obtenues. Dans ce chapitre, on retrouve une multitude de tableaux et graphiques qui permettent de voir les résultats expérimentaux que nous avons obtenu. Une analyse détaillée de ces derniers y est aussi présente.

Le chapitre 6 énumère finalement les conclusions qui ressortent du travail de recherche. On y trouve un récapitulatif des différents résultats d'expérimentation ainsi que les conclusions qui en découlent. Une description de possibles travaux futurs y est aussi fournie.

CHAPITRE 2

REVUE DE LITTÉRATURE

Dans ce chapitre nous présentons tout d'abord une description informelle de l'analyse syntaxique. On y aborde principalement sa définition ainsi que les concepts de base qui y sont rattachés. Plus précisément, l'algorithmique derrière l'analyse syntaxique et la grammaire hors contexte qui y est intimement liée. Puis, une description plus poussée et formelle de l'algorithme d'Earley est présentée, élément qui est au coeur des techniques d'analyse syntaxique robuste choisies.

Ensuite, un recensement des algorithmes d'analyse syntaxique robuste les plus connus et utilisés est effectué. Une attention particulière est portée sur les algorithmes pouvant fonctionner avec toutes les classes de grammaires hors contexte ainsi que ceux étendant l'algorithme d'Earley pour offrir une tolérance aux erreurs. Plus particulièrement les trois grandes familles d'algorithmes suivantes : correction locale, régionale et globale des erreurs.

2.1 Analyse syntaxique

L'analyse syntaxique est un domaine de l'informatique très bien compris et qui a été grandement étudié au cours des 50 dernières années. C'est au courant des années 70 que les principes théoriques de l'analyse syntaxique ont été couchés sur papier par des scientifiques de renoms tels que Daniel E. Knuth, Alfred Aho, Jeffrey Ullman ainsi que plusieurs autres.

Au sens large, la définition de l'analyse syntaxique peut se lire comme suit : « processus permettant de structurer une représentation linéaire en accord avec une grammaire donnée » (Grune et Jacobs, 1990, Traduction libre - Chapitre 1). Cette définition hautement abstraite ne l'est pas sans raison. En effet, l'analyse syntaxique s'applique à des domaines très variés tels que le traitement de texte, la musique, la géologie. Il est donc impératif que sa définition englobe tous les champs d'expertise auxquels elle peut servir. Le présent projet de recherche se concentre principalement sur l'analyse syntaxique d'un ensemble restreint de phrases textuelles. Pour offrir une définition efficace de l'analyse syntaxique dans le contexte du projet, il est important de décortiquer les différents éléments de la définition : la structure, la grammaire, la représentation linéaire et finalement le processus.

La structure d'une phrase dans le contexte de ce travail est définie comme étant le regroupement d'un ou plusieurs mots en constituants, eux-mêmes agrégés en constituants plus gros afin d'atteindre un élément final (la racine) englobant tous les mots d'une phrase. Le grou-

pement des mots en constituants s'effectue en respectant les productions de la grammaire, notion qui sera approfondie dans les prochains paragraphes. Une représentation visuelle de la structure est souvent donnée sous forme d'arbre, car cette structure permet de discerner rapidement les différents constituants de la phrase. Dans cette représentation de la structure, les noeuds de l'arbre forment les constituants tandis que chacune des feuilles représente un mot dans la phrase. C'est ce qu'on appelle communément un arbre syntaxique.

La grammaire est à la structure d'une phrase ce qu'un plan est à la charpente d'un bâtiment : c'est elle qui détermine les manières possibles de regrouper les mots en constituants. La grammaire permet de limiter le dialogue à un ensemble prédéterminé de phrases spécifiques, par exemple, l'ensemble des phrases de la langue française. Cet ensemble de phrases peut être fini ou infini en fonction des besoins de l'application utilisant la grammaire. C'est le développeur de la grammaire qui est responsable de limiter les phrases à certaines constructions prédéterminées ainsi qu'à certains mots minutieusement choisis. Plusieurs dizaines de formalismes de grammaires différents existent, chacun d'eux essayant de régler différents problèmes. Voir (Grune et Jacobs, 1990, Chapitre 2) pour une classification des grammaires utilisées lors de l'analyse syntaxique des phrases.

Dans le cadre particulier de notre projet de recherche, nous avons concentré nos efforts sur les grammaires hors contexte. Ces grammaires sont définies par un ensemble de terminaux (les mots) et de non-terminaux reliés entre eux grâce à une série de règles appelées couramment productions. Chaque constituant représente une séquence de terminaux et non-terminaux. Les productions permettent de combiner des constituants d'une manière stricte afin de créer un nouveau constituant. Les productions peuvent être vues simplement comme les règles permettant de structurer une phrase.

Formellement, une grammaire hors contexte est formée d'un tuple de quatre éléments : $G = (N, T, P, S)$. L'ensemble N contient les non-terminaux de la grammaire. L'ensemble T, disjoint de N, contient les terminaux de la grammaire, soit l'ensemble des mots pouvant être reconnus. La variable P est un ensemble contenant les productions de la grammaire où chaque production est une relation de N vers $(N \cup T)^*$. Les productions sont de la forme $A \rightarrow \alpha$. La partie à gauche de la flèche, A, est appelée le non-terminal de la production et fait partie de l'ensemble N. La partie à droite de la flèche, α, est appelée l'expansion de la production. L'ordre dans une production définit l'ordre dans lequel les mots ou constituants peuvent apparaître. Finalement, la variable S définit le symbole de départ de la grammaire et existe dans l'ensemble N.

Dans le but d'alléger la définition des grammaires hors contexte, on a souvent recours à la barre verticale pour dénoter une alternative dans une production. Par exemple, au lieu d'écrire trois productions distinctes ayant le même non-terminal, on utilise plutôt la barre

verticale (|) pour fusionner les trois productions en une seule. Les définitions 2.1 et 2.2 reconnaissent exactement le même langage et sont du même coup totalement équivalentes.

$$\begin{aligned} A &\rightarrow \textbf{oui} \\ A &\rightarrow \textbf{non} \\ A &\rightarrow \textbf{possiblement} \end{aligned} \tag{2.1}$$

$$A \rightarrow \textbf{oui} \mid \textbf{non} \mid \textbf{possiblement} \tag{2.2}$$

Le formalisme des grammaires hors contexte définit également le symbole ϵ pour représenter la chaîne vide. Il sert habituellement à créer des productions permettant de définir les constituants optionnels d'un langage. Lorsqu'on conçoit un langage à l'aide d'une grammaire hors contexte, il arrive souvent qu'on ait besoin de reconnaitre optionnellement un sous-ensemble de mots. Par exemple, les adjectifs d'une phrase sont optionnels dans la langue française. Pour parvenir à modéliser un tel phénomène, on utilise alors le symbole ϵ qui dénote la création d'une production vide. Si une production doit reconnaitre **oui**, **non** ou rien du tout, alors on la définira de cette manière : $A \rightarrow$ **oui** | **non** | ϵ. L'expansion d'une production peut donc être composée soit d'une séquence de terminaux ou non-terminaux ou de la production vide. Plus formellement, pour la production $A \rightarrow \alpha$, $\alpha \in \{(N \cup T)^*\} \cup \{\epsilon\}$.

La représentation linéaire dont il est question dans la définition de l'analyse syntaxique n'est qu'un simple terme faisant référence à une phrase écrite. En effet, une phrase est une représentation linéaire d'un arbre syntaxique et qui est lui même une représentation structurée d'une phrase.

Arrêtons-nous ici afin d'illustrer les précédents concepts à l'aide d'une analogie. Concentrons-nous sur une analogie se rapprochant du projet de recherche : l'ensemble des phrases de la langue française ayant dans l'ordre un sujet, un adjectif, un verbe et un nom précédé d'une préposition. Nous allons restreindre chaque constituant à certains mots. Les sujets peuvent être **dominique** ou **michel**, les adjectifs **beau** ou **grand**, les verbes pourront être **voit** ou **boit** et finalement les noms qui peuvent être précédés d'une préposition seront **du jus** et **de l'eau**. Avec ces informations, nous sommes maintenant en mesure de définir une grammaire qui permettra de structurer une représentation linéaire (une phrase) à l'aide d'un processus qui est encore à définir. La définition 2.3 présente la grammaire associée à ce langage.

$$
\begin{aligned}
P^* &\rightarrow \textbf{le } A\ S\ V\ N \\
A &\rightarrow \textbf{beau} \mid \textbf{grand} \\
S &\rightarrow \textbf{michel} \mid \textbf{dominique} \\
V &\rightarrow \textbf{boit} \mid \textbf{voit} \\
N &\rightarrow \textbf{du jus} \mid \textbf{de l'eau}
\end{aligned} \tag{2.3}
$$

La grammaire est exprimée à l'aide d'une forme simplifiée de Backus-Naur. Chaque ligne indique une production de la grammaire et chaque production peut être composée de plus d'une alternative. Les lignes forment l'ensemble P de la définition formelle. Les lettres majuscules dénotent les non-terminaux, l'ensemble N, tandis que les mots en gras représentent les terminaux, l'ensemble T. La flèche indique que le non-terminal à gauche peut générer la séquence à droite soit l'expansion de la production. Un astérisque suivant le non-terminal d'une production définit le non-terminal de départ de la grammaire. Cette production doit être reconnue par l'analyseur syntaxique pour avoir une phrase bien formée. La production $N \rightarrow$ **du jus** | **de l'eau** nous indique que le non-terminal N peut générer la séquence **du jus** ou la séquence **de l'eau**. Cette syntaxe est utilisée pour la définition des grammaires et est celle qui sera employée tout au long de ce mémoire.

En utilisant la définition de cette grammaire, on peut, à l'aide d'un processus, structurer une représentation linéaire. Prenons la phrase **le beau michel boit du jus** comme représentation linéaire. Ce processus utilise la grammaire et détermine si la phrase fait partie de l'ensemble des phrases bien formées. Dans le cas où la phrase est bien formée, un arbre de syntaxe pourra être produit par le processus. La figure 2.1 permet de visualiser l'arbre de syntaxe résultant du processus de reconnaissance de la phrase.

Le processus est simplement un algorithme prenant en argument une grammaire, hors contexte dans le cas qui nous concerne, ainsi qu'une phrase. Le processus retourne ensuite en sortie un arbre syntaxique si la phrase peut être générée par la grammaire. Si elle ne peut être reconnue par la grammaire, le processus ne retourne rien.

Même si la principe est simple, l'algorithme sous-jacent l'est beaucoup moins. Au cours des dernières années, des dizaines d'algorithmes différents ont été proposés, certains plus efficaces que d'autres en terme de complexité, de compréhension, de performance, de consommation de mémoire ou d'une combinaison des éléments précédents. C'est donc dire que l'éventail d'algorithmes est plutôt vaste et que certains critères doivent être définis afin d'arrêter sa décision sur un algorithme en particulier.

Dans notre projet de recherche, les grammaires ne sont pas connues d'avance et ne sont pas fixes, elles évoluent au fur et à mesure du développement d'un projet. La principale contrainte est donc de pouvoir faire l'analyse syntaxique à l'aide de n'importe quelle grammaire hors

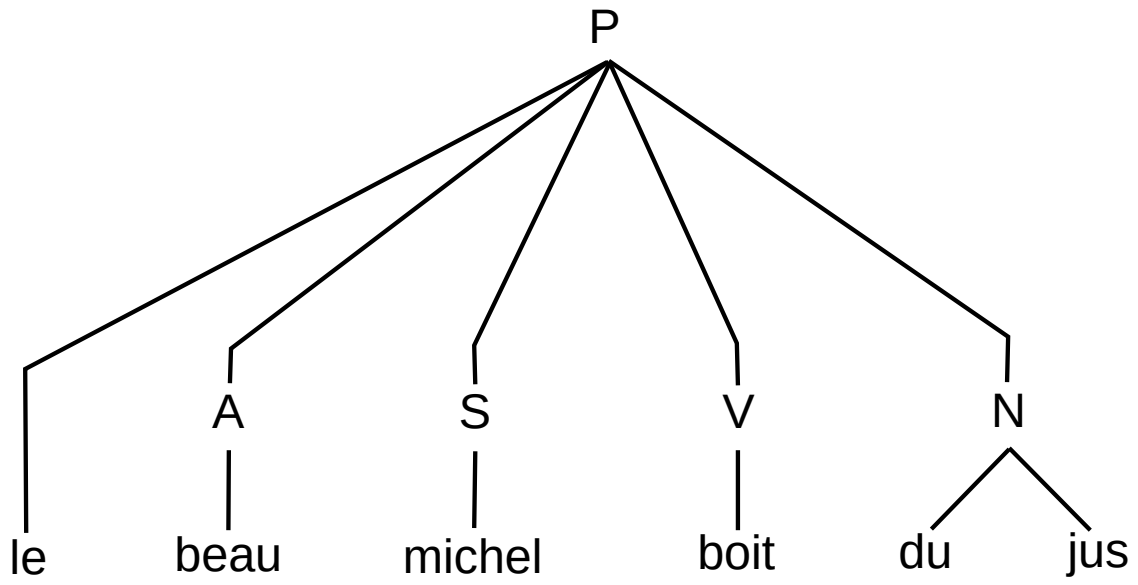

Figure 2.1 L'arbre de syntaxe pour la phrase **le beau michel boit du jus**

contexte. Il n'est donc pas possible d'utiliser des algorithmes qui ont des restrictions quant à la forme de la grammaire hors contexte qu'ils peuvent accepter.

Avec ce critère, le choix d'un algorithme d'analyse syntaxique se réduit considérablement. Pour le projet de recherche, nous concentrerons nos efforts sur l'algorithme d'Earley. C'est un algorithme efficace qui fonctionne avec toutes les grammaires hors contexte. C'est également un algorithme simple à comprendre et à implémenter. Il s'exécute avec une complexité maximale de $O(n^3)$, $O(n^2)$ pour les grammaires non ambiguës et $O(n)$ dans certains cas. La variable n représente le nombre de mots dans la phrase. Voir (Earley, 1970) pour plus de détails sur la complexité de l'algorithme. Il est particulièrement souple, ce qui permet d'étendre ses capacités au-delà de la simple analyse syntaxique habituelle. L'algorithme est également utilisé en production par l'entreprise partenaire. Tous ces éléments nous ont donc poussé vers cet algorithme comme base pour l'analyse syntaxique robuste.

Puisque cet algorithme est à la base des algorithmes d'analyse syntaxique robuste qui seront évalués lors de cette recherche, la prochaine section expliquera plus en détails le fonctionnement de l'algorithme d'Earley.

Mais avant de continuer, arrêtons-nous quelques instants pour donner une description rapide des différents éléments de notation utilisés tout au long de ce mémoire, particulièrement en ce qui a trait aux grammaires et aux algorithmes. Les lettres majuscules commençant au début de l'aphabet sont utilisées pour dénoter les non-terminaux d'une production (A, B, C, ...). Les terminaux sont représentés par des mots en minuscules gras (**plusieurs**, **mots**,

...). Les lettres grecques (α, β, γ, ...) sont utilisées pour dénoter une séquence, possiblement vide, d'un ou plusieurs symboles, un symbole pouvant être un non-terminal ou un terminal. Les lettres et mots en italiques représentent des variables. Le tableau 2.1 recense les variables fréquemment utilisées dans ce mémoire ainsi que leur signification.

Tableau 2.1 Liste des symboles fréquemment utilisés dans le cadre de ce mémoire

Variable	Sens
N	L'ensemble des non-terminaux d'une grammaire
T	L'ensemble des terminaux d'une grammaire
P	L'ensemble des productions d'une grammaire
S	Le symbole de départ de la grammaire
s_0, s_1, ...	Représente un ensemble d'items d'Earley
t_0, t_1, ...	Représente un mot dans une phrase
t	Représente un mot quelconque
n	Nombre de mots dans un phrase
e	Compteur d'erreurs

2.1.1 Algorithme d'Earley

Cet algorithme d'analyse syntaxique général a été développé par Jay Earley à la fin des années 60 dans le cadre de sa thèse de doctorat. L'article relatant la technique (Earley, 1970) a été publié pour la première fois dans le journal « *Communications of the ACM* » sous le titre « *An Efficient Context-Free Parsing Algorithm* ». La présente section donnera une description de l'algorithme décrit dans l'article : les entrées requises, les structures de données qu'il utilise ainsi que les étapes nécessaires à son implémentation.

Tout d'abord, l'algorithme reçoit en entrée deux paramètres : une grammaire hors contexte notée *grammar* et une liste de mots représentant la phrase. La phrase reçue en entrée est une phrase textuelle où chaque mot est séparé par un espace. Chaque mot est numéroté $t_0, t_1, \ldots t_{n-1}$. On retrouve donc n mots dans la phrase.

La structure principale la plus importante de l'algorithme est l'item d'Earley. Cet item s'inspire fortement des items LR inventés par D. E. Knuth dans le cadre du développement de l'algorithme d'analyse syntaxique LR(k) (Knuth, 1965). Un item d'Earley est un tuple composé de quatre éléments : une production p de la grammaire tel que $p \in P$, un point déterminant la position actuelle dans la partie droite de la production appélé *dotOffset*, un index nommé *currentIndex* déterminant où se trouve actuellement l'item par rapport à la liste de mots et finalement un index, désigné par *earlierIndex* déterminant où a commencé l'item

par rapport à la liste de mots. Pour faciliter la lecture, les items d'Earley sont représenté de la manière suivante :

$$[A \rightarrow \alpha \bullet \beta, f, c]$$

Si on décortique l'item affiché precédemment, $A \rightarrow \alpha\ \beta$ est la production (p) de la grammaire où α et β sont des séquences, possiblement vides, de terminaux et non-terminaux ($\alpha, \beta \in (N\ \cup\ T)^*$). Le point ($\bullet$) représente le *dotOffset* et est contraint par $0 \leq dotOffset \leq \|p\|$ où $\|p\|$ représente le nombre de symboles dans l'expansion de la production. La valeur c représente le *currentIndex* et la valeur f le *earlierIndex*. Ces deux dernières valeurs sont contraintes par $0 \leq f \leq c \leq n$.

La deuxième structure d'importance de l'algorithme est l'ensemble d'Earley. Ce dernier est simplement un ensemble contenant tous les items d'Earley à un certain point de l'analyse. Si on aligne les mots horizontalement, il y aura un ensemble avant et après chaque mot. Ceci implique donc qu'il y a $n + 1$ ensemble d'Earley lors de l'analyse d'une phrase, où n est le nombre de mots. Les ensembles sont numérotés $s_0, s_1, \ldots, s_n$. L'ensemble d'Earley respecte les propriétés des ensembles et ne contiendra donc jamais deux items identiques pour un même ensemble s_i. Les ensembles d'Earley sont peuplés en ligne, c'est-à-dire au fur et à mesure qu'on analyse les mots. Si un seul mot a été analysé sur les n, alors il y aura au maximum deux ensembles peuplés. La figure 2.2 illustre la façon dont sont disposés les ensembles par rapport aux mots.

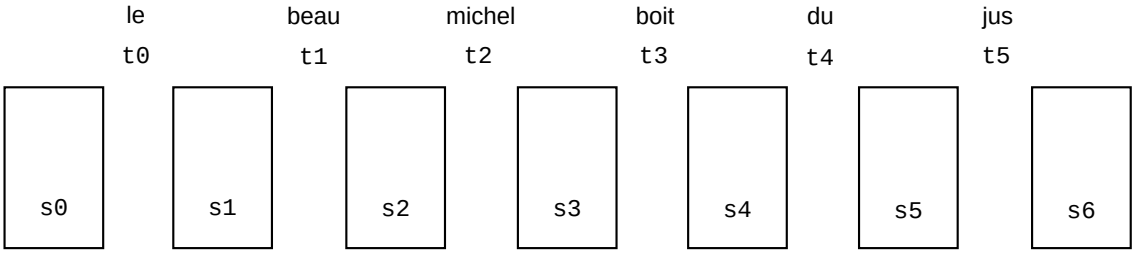

Figure 2.2 Dispositions des ensembles d'Earley par rapport aux mots

Nous entamons maintenant la partie traitant du fonctionnement de l'algorithme d'Earley. Ce dernier est basé sur quatre fonctions principales : la fonction *scan* (2.3), la fonction *perdict* (2.4), la fonction *complete* (2.5) et la fonction *recognize* (2.1) qui est le point d'entré de l'algorithme.

Il est très aisé de faire une analogie entre l'algorithme et l'arbre syntaxique. En effet, on peut voir les items d'Earley comme des arbres où le *dotOffset* représente où est rendue la complétion du sous-arbre ayant comme racine le non-terminal de la production associée à

l'item. Le non-terminal d'un item représente la racine d'un sous-arbre tandis que l'expansion représente ses enfants. Un terminal dans l'expansion d'une production représente une feuille et un non-terminal définit un autre sous-arbre. Le but de l'algorithme est donc de construire un arbre où la racine est le symbole de départ de la grammaire. Les feuilles et les noeuds de cet arbre représentent la façon dont la phrase est structurée. La figure 2.3 démontre cette analogie entre les items et les arbres. Elle contient deux items, un complet et un incomplet. Le premier item $[A \rightarrow \textbf{beau} \bullet, 1, 2]$ est un sous-arbre requis par le deuxième item $[P \rightarrow \textbf{le}\ A\ \bullet\ S\ V\ N, 0, 2]$, il est donc inséré au bon emplacement dans le deuxième arbre, qui est l'arbre final que l'algorithme cherche à compléter.

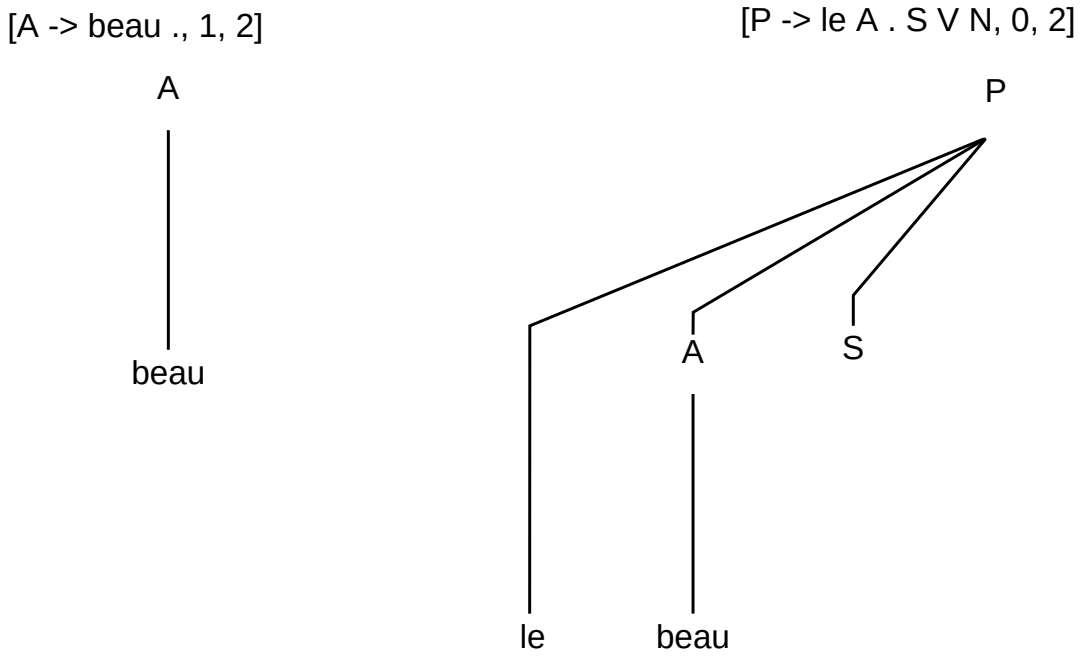

Figure 2.3 Analogie entre les items d'Earley et les arbres

En gardant en mémoire la figure 2.3, nous allons dès à présent décrire le travail accompli par chacune des quatre fonctions importantes de l'algorithme d'Earley.

Avant de commencer la description de l'algorithme, nous introduisons le symbole $\dashv$ qui représente la fin d'une phrase. Quand ce symbole est accepté par l'algorithme dans une phrase, cela implique que l'analyse est terminée. Il ne peut donc pas y avoir de terminaux après le symbole $\dashv$. Ce symbole ne doit pas être présent préalablement dans T. On considère ce symbole comme étant un terminal, il fait donc partie de l'ensemble T. La grammaire est modifiée en conséquence en augmentant la définition de $T : T \leftarrow T \cup \{\dashv\}$. Cela veut également dire que la phrase reçue par l'algorithme doit impérativement se terminer par le terminal $\dashv$. Pour respecter cette assertion, l'algorithme s'assure que le dernier mot d'une phrase soit $\dashv$.

Si ce n'est pas le cas, l'algorithme l'ajoute automatiquement.

La fonction principale *recognize* (2.1) est le point d'entrée de l'algorithme et s'occupe d'effectuer toutes les étapes intermédiaires. Au début de cette fonction, $n + 1$ ensembles d'Earley sont initialisés. La production $Z \rightarrow S \dashv$ est ajoutée à la grammaire : $P \leftarrow P \cup \{Z \rightarrow S \dashv\}$. Cette nouvelle production est le point de départ de l'algorithme. L'item $[Z \rightarrow \bullet\ S\ \dashv, 0, 0]$ est ensuite ajouté à l'ensemble s_0, c'est l'item d'Earley de départ. Après ce travail d'initialisation, l'algorithme effectue n itérations, de $i = 0$ jusqu'à n et pour chaque itération, la fonction *computeSet* est appelée. La variable i est accessible aux autres fonctions de l'algorithme et permet de déterminer le mot courant t_i, l'ensemble d'Earley courant s_i ainsi que le prochain ensemble d'Earley s_{i+1} dans les cas où $i \neq n$.

Algorithme 2.1 Fonction *recognize* de l'algorithme d'Earley

pour $i = 0$ **à** n **faire**
    $s_i \leftarrow$ EarleySet()

$P \leftarrow P \cup \{Z \rightarrow S \dashv\}$
$s_0 \leftarrow s_0 \cup \{[Z \rightarrow \bullet\ S \dashv, 0, 0]\}$

pour $i = 0$ **à** $n - 1$ **faire**
    computeSet()

À chaque itération, la fonction *computeSet* (2.2) fait appel aux trois autres fonctions principales afin de peupler l'ensemble s_i ainsi que l'ensemble s_{i+1} s'il est défini. La fonction appelle en premier lieu la fonction *scan*. Puisque cette fonction peuple l'ensemble s_{i+1}, elle est appelée seulement dans les cas où il est défini, soit quand $i \neq n$. Ensuite, elle effectue la phase de prédiction en appelant *predict* puis la phase de complétion à l'aide de la fonction *complete*. Cette série d'appels est repétée tant et aussi longtemps que s_i ou s_{i+1} a changé.

La fonction *scan* (2.3) est responsable d'incrémenter le *dotOffset* des items qui sont bloqués à un terminal si ce terminal correspond au mot courant t_i. D'un point de vue formel, la fonction cherche dans l'ensemble s_i tous les items de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, i]$ tel que $\mathbf{t} = t_i$, où α et β sont des séquences de symboles possiblement vides. Pour chaque item trouvé, la fonction ajoute l'item $[A \rightarrow \alpha\ \mathbf{t} \bullet \beta, f, i + 1]$ à l'ensemble s_{i+1}.

Du point de vue des arbres, la fonction s'occupe d'ajouter une feuille (ayant comme valeur t_i) à tous les items trouvés ayant la bonne forme. La figure 2.4 illustre le travail fait par une étape de la phase de scan.

La fonction *predict* (2.4) a quant à elle la responsabilité d'ajouter dans s_i les items permettant de débloquer les items bloqués, c'est-à-dire que le point est juste avant un non-terminal.

Algorithme 2.2 Fonction *computeSet* de l'algorithme d'Earley

repéter
  $s'_i = s_i$
  $s'_{i+1} = s_{i+1}$

  si $i \neq n$ **alors**
    scan()

  predict()
  complete()
jusqu'à $s'_i = s_i$ **et** $s'_{i+1} = s_{i+1}$ // s_i, s_{i+1} possiblement modifiés par une des fonctions

Algorithme 2.3 Fonction *scan* de l'algorithme d'Earley

pour tout item de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\beta, f, i]$ **dans** s_i **faire**
  si $t_i = \mathbf{t}$ **alors**
    $s_{i+1} \leftarrow s_{i+1} \cup \{[A \rightarrow \alpha\ \mathbf{t} \bullet \beta, f, i+1]\}$

On peut voir cette fonction comme étant celle qui crée de nouveaux items d'Earley afin de faire avancer l'analyse. Cette fonction trouve tous les items de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, i]$ dans l'ensemble s_i. Pour chacun de ces items, elle cherche les productions de la grammaire telles que le non-terminal de la production est B. Pour chaque production trouvée, la fonction ajoute l'item $[B \rightarrow \bullet\ \gamma, i, i]$ à l'ensemble s_i.

Algorithme 2.4 Fonction *predict* de l'algorithme d'Earley

pour tout item de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, i]$ **dans** s_i **faire**
  si $B \rightarrow \epsilon \in P$ **alors**
    $s_i \leftarrow s_i \cup \{[A \rightarrow \alpha B \bullet \beta, f, i]\}$

  pour tout production de la forme $B \rightarrow \gamma$ **tel que** $\gamma \neq \epsilon$ **dans** P **faire**
    $s_i \leftarrow s_i \cup \{[B \rightarrow \bullet\ \gamma, i, i]\}$

La fonction *predict* permet de démarrer de nouveaux sous-arbres qui sont requis par des arbres présentement bloqués à un non-terminal. La figure 2.5 montre le travail fait par une étape de la phase de complétion.

La dernière fonction, la fonction *complete* (2.5), est chargée de débloquer un item dont le *dotOffset* précède un non-terminal. La fonction cherche dans l'ensemble s_i tous les items de la forme $[B \rightarrow \gamma\ \bullet, j, i]$. Ce sont les items qui sont complétés dans l'ensemble s_i. Pour

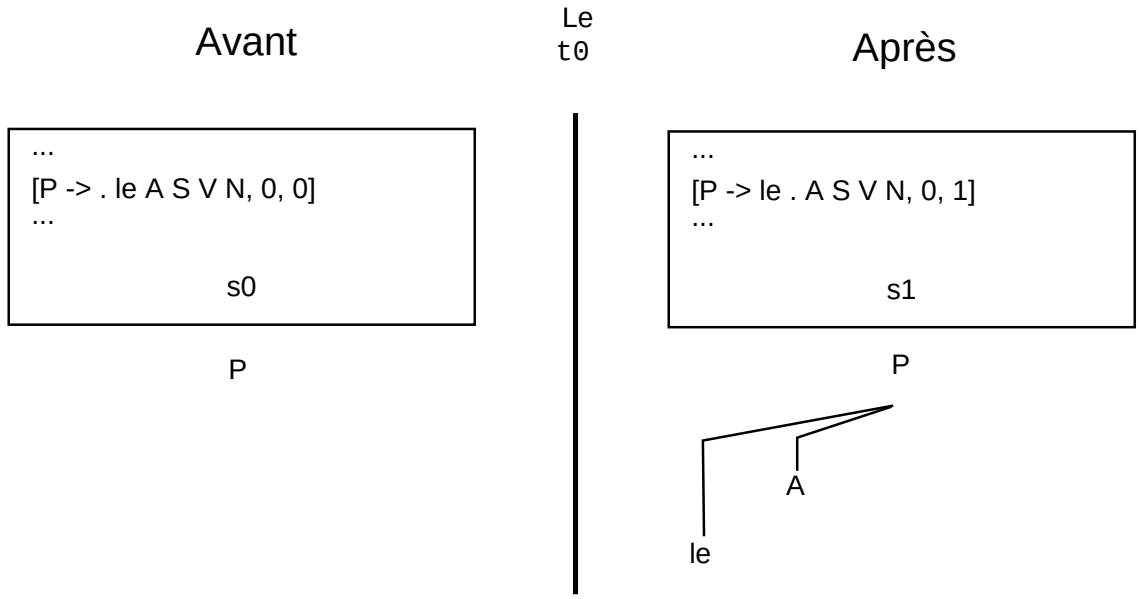

Figure 2.4 Démonstration du travail fait par une étape de scan

chaque item complété, la fonction recherche ensuite dans l'ensemble s_j tous les items de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, j]$. Ce sont les items bloqués à un non-terminal pour lesquelles il existe un item complétant le non-terminal. Pour chacun des items, l'algorithme ajoute l'item $[A \rightarrow \alpha\ B \bullet \beta, f, i]$ à l'ensemble s_i.

Algorithme 2.5 Fonction *complete* de l'algorithme d'Earley

pour tout item de la forme $[B \rightarrow \gamma \bullet, j, i]$ **dans** s_i **faire**
  pour tout item de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, j]$ **dans** s_j **faire**
    $s_i \leftarrow s_i \cup \{[A \rightarrow \alpha\ B \bullet \beta, f, i]\}$

Du point de vue de l'analogie avec les arbres, la fonction *complete* est responsable d'agréger un sous-arbre avec un autre qui a besoin de ce dernier pour continuer sa construction. La figure 2.6 montre le travail fait par une étape de la phase de complétion.

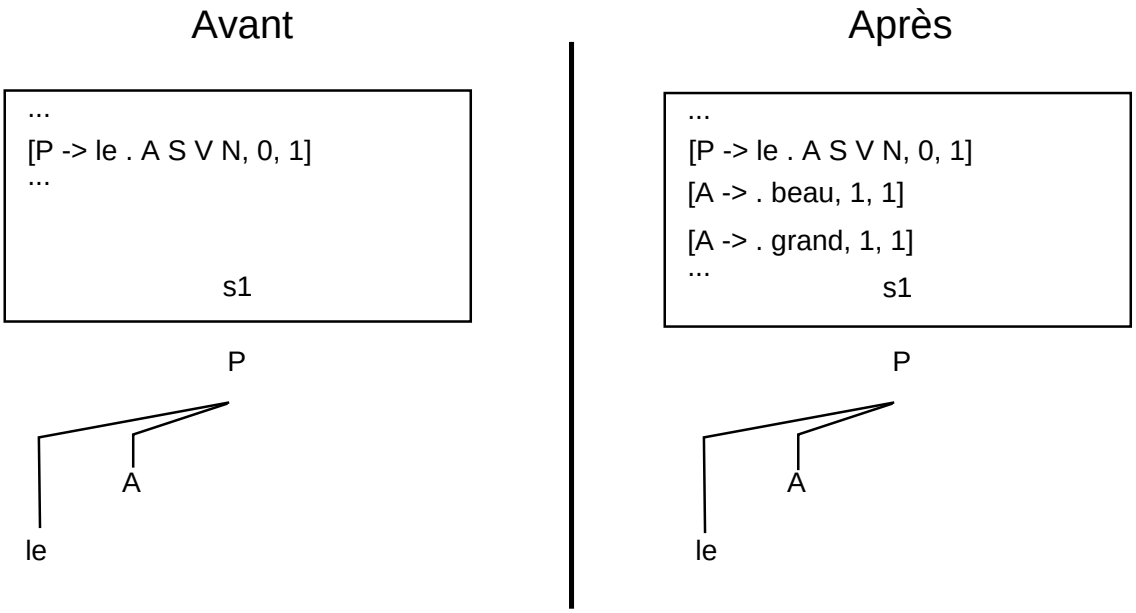

Figure 2.5 Démonstration du travail effectué par une étape de prédiction

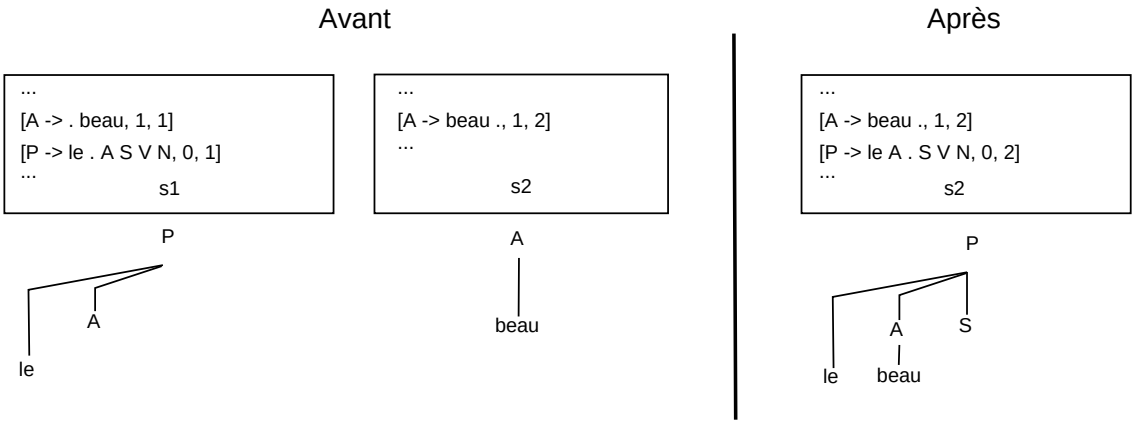

Figure 2.6 Démonstration du travail effectué par une étape de complétion

2.2 Analyse syntaxique robuste

L'analyse syntaxique robuste est une branche de l'analyse syntaxique classique. Un algorithme d'analyse syntaxique robuste est définit comme un algorithme syntaxique qui est en mesure de déterminer la structure syntaxique d'une représentation linéaire contenant une ou plusieurs formes d'erreurs. De plus, dans notre cadre précis, nous ajoutons également que l'algorithme doit être en mesure d'effectuer la correction de ces erreurs. Ceci veut donc dire qu'une version corrigée de la phrase originale sera disponible lorsque l'algorithme d'analyse syntaxique robuste aura effectué son analyse.

Le terme erreur est plutôt vaste et dépend de la représentation linéaire. Puisque nous travaillons avec une phrase textuelle comme représentation linéaire, c'est dans cette optique que nous décrivons le processus d'analyse syntaxique robuste. On parlera alors du terme erreur comme étant l'ensemble des erreurs pouvant affecter une phrase : les erreurs lexicales, syntaxiques, sémantiques, logiques ou une combinaison des quatre. Notre projet de recherche étant focalisé sur le diagnostic des erreurs syntaxiques, nous traiterons seulement du cas de cette classe d'erreurs.

Depuis les débuts de l'analyse syntaxique, la robustesse face aux erreurs a reçu une grande attention. En effet, et plus particulièrement de nos jours, il est de plus en plus courant de devoir analyser de larges quantités d'informations de diverses provenances. La majorité du temps, cette masse d'information contient une multitude d'erreurs syntaxiques très variées allant de l'oubli d'un mot à l'inversion de constituants. Là où les algorithmes d'analyse syntaxique classiques sont démunis, les algorithmes syntaxiques robuste prennent la relève.

Durant les 40 dernières années, plusieurs dizaines de techniques ont été explorées. Certaines de ces techniques utilisent les grammaires hors contexte tandis que d'autres se concentrent sur des formalismes de grammaire complètement différents. Bien que le présent projet tourne exclusivement autour des grammaires hors contexte, nous référons tout de même le lecteur vers des techniques d'analyse syntaxique robuste faisant appel à d'autres classes de grammaire. En effet, les idées derrière ces techniques servent à mieux comprendre le processus de l'analyse syntaxique robuste. Les prochains paragraphes sont dédiés à de telles techniques.

Un article (Weischedel et Sondheimer, 1983) propose d'utiliser un réseau de transition augmenté (*Augmented Transition Network* (ATN)) pour effectuer une analyse syntaxique robuste aux erreurs. Un ATN est constitué de noeuds et d'arcs et chaque arc est associé à une condition. Il y a un noeud de départ et un noeud final. Pour passer d'un noeud à un autre, la condition de l'arc reliant les deux noeuds doit être respectée. On reconnait une phrase si on peut commencer au noeud de départ et suivre un chemin jusqu'au noeud final, où chaque passage d'arc permet de reconnaitre un mot de la phrase. Les auteurs proposent

l'idée d'ajouter des méta-règles, ou *meta-rules* en anglais, aux arcs lorsque leurs conditions ne sont pas atteintes afin de rendre l'ATN robuste aux erreurs. Les nouvelles règles sont attachées aux conditions déjà présentes. Elles sont choisies avec soin afin de ne cibler qu'un seul type d'erreur à la fois et ainsi empêcher du même coup une surpopulation de conditions. Ces méta-règles permettent alors de relaxer les conditions de base afin d'accepter des entrées comportant des erreurs.

Un autre article intéressant (Heinecke *et al.*, 1998) décrit une technique travaillant avec des grammaires de contraintes, ou *Constraint Grammar* (CG) en anglais. Ce formalisme de grammaire a été développé par Karlsson en 1990 (Karlsson, 1990). La grammaire est définie à l'aide de contraintes qui sont, de manière formelle, un tuple de quatre éléments : le domaine, la cible, l'opérateur et les conditions de contexte. Les auteurs utilisent un algorithme de satisfaction de contraintes afin d'effectuer l'analyse syntaxique d'une phrase. Sans modifications, un tel algorithme doit respecter toutes les contraintes pour qu'une phrase soit reconnue. Ce comportement empêche la réussite de l'analyse d'une phrase qui viole une ou plusieurs contraintes. Pour pallier ce problème, les auteurs élaborent la notion de contraintes graduées. Ces contraintes graduées possèdent un poids normalisé entre 0.0 et 1.0. Une contrainte avec un poids de 0.0 ne peut être violée tandis qu'une contrainte près de 1.0 est plutôt une préférence qu'une obligation. Les contraintes possédant un poids de 1.0 sont obligatoires. Si toutes les contraintes ont un poids de 1.0, cela revient à l'algorithme sans relaxation. Ce poids permet alors à l'algorithme de devenir robuste aux erreurs en relâchant des contraintes. Cette relaxation est la clé afin de trouver une ou plusieurs solutions même dans les cas où des contraintes ne seraient pas entièrement satisfaites et permet donc de rendre le processus d'analyse robuste aux erreurs.

Un article récent (Vanrullen *et al.*, 2006) décrit une technique de correction d'erreur utilisant des grammaires de propriétés, ou *Property Grammar* (PG) en anglais. Ce formalisme de grammaires est très semblable aux grammaires CG puisque les constituants de la grammaire sont définis à l'aide de contraintes. Ces contraintes, appelées propriétés dans les grammaires PG, permettent d'établir la structure syntaxique d'une phrase. Ces propriétés se distinguent des contraintes des grammaires CG par la manière dont elles sont définies. Habituellement, les erreurs contenues dans une phrase se retrouvent seulement à deux ou trois emplacements dans celle-ci. Il y a donc de grandes parties de phrases qui sont bien formées. Ces parties de phrases rencontrent certaines propriétés de la grammaire PG. Ces propriétés satisfaites permettent aisément d'obtenir une analyse partielle des différents bouts de la phrase. L'idée ici est de permettre à certaines propriétés d'être relaxées. Cette relaxation survient lorsqu'une propriété n'est pas rencontrée. Dans ce cas, on accepte tout de même l'analyse en notant la propriété qui a été violée. Cette relaxation permet alors de lier plusieurs analyses par-

tielles valides ensemble et d'obtenir une analyse complète. L'espace de recherche est réduit en recherchant les analyses complètes brisant le moins de propriétés. La robustesse aux erreurs provient donc de la capacité de l'algorithme de lier ensemble des parties bien formées disjointes.

Dans les articles mentionnés ci-dessus, une notion commune est mise en évidence : la relaxation de contraintes. Cette idée revient en effet dans la plupart des algorithmes d'analyse syntaxique robuste développés à ce jour. Ceci est tout à fait normal, car l'analyse syntaxique classique est d'une certaine façon rigide : la phrase respecte la grammaire ou non. Pour devenir robuste, il faut alors trouver un moyen de permettre à l'analyse de continuer lorsque certaines propriétés sont violées à cause d'une ou plusieurs erreurs dans la phrase. Les algorithmes d'analyse syntaxique robuste se distinguent entre eux par la manière dont le relâchement des contraintes s'effectue et par leur habileté à limiter l'espace de recherche. Cette habileté est d'autant plus importante que la relaxation des contraintes amène bien souvent une explosion combinatoire des possibilités.

Dans le cadre de notre projet de recherche, être capable d'effectuer l'analyse syntaxique avec l'ensemble des grammaires hors contexte est un requis fondamental. Les algorithmes choisis ne doivent donc pas se limiter à certaines classes de grammaires hors contexte comme les algorithmes LR(k) [1] et LL(k) [2]. Les algorithmes qui ne sont pas restreints par une classe de grammaire particulière sont dits généraux. La grande majorité des techniques les plus connues pouvant traiter n'importe quelle grammaire hors contexte comme CYK (Cocke, 1969), (Younger, 1966), (Kasami, 1965), Earley (Earley, 1970) et *Generalized Left-to-right, Rightmost derivation* (GLR) (Tomita, 1985) ont été améliorées par différents chercheurs dans le but d'en créer des versions robustes aux erreurs. Certaines de ces versions ne sont que robuste aux erreurs tandis que d'autres effectuent également la correction, ce qui permet d'établir un diagnostic.

La suite de ce chapitre est séparée en trois sous-sections, une pour chaque grande famille d'algorithmes d'analyse syntaxique robustes. Ces trois grandes familles sont, dans l'ordre de présentation respective : les algorithmes effectuant une correction locale, régionale et globale des erreurs.

2.2.1 Correction d'erreur locale

Dans cette sous-section, nous décrivons quelques algorithmes d'analyse syntaxique robustes effectuant une correction locale des erreurs. Le terme *local* fait référence au fait que les algorithmes corrigent les erreurs en tenant compte seulement de ce qui s'en vient dans

1. Voir (Knuth, 1965) pour plus d'information
2. Voir (Rosenkrantz et Stearns, 1969) pour plus d'information

la phrase et non ce qui a déjà été analysé. Par exemple, pour la phrase **le beau michel boit du jus**, si l'algorithme a déjà pris des décisions pour la partie **le beau michel**, les prochaines décisions ne se prendront qu'en tenant compte de ce qui suit cette partie, soit **boit du jus**, la partie **le beau michel** étant considérée comme correcte. Ces algorithmes se rapprochent donc grandement de la technique algorithmique dite vorace (*greedy* en anglais), car ils prennent la décision la plus prometteuse sans tenir compte des décisions précédentes. Ils modifient généralement un ou plusieurs mots restant à analyser de manière à permettre à l'algorithme de continuer son travail pour au moins un autre mot.

Ces algorithmes ont l'avantage d'être plus rapides que leurs homologues effectuant une correction globale ou régionale, car l'espace de recherche requis pour trouver une analyse complète est grandement réduit. Cependant, comme ils n'explorent pas tous les chemins, ils ont le désavantage de proposer des solutions parfois moins optimales dans certains cas.

L'algorithme d'Anderson-Backhouse (Anderson et Backhouse, 1981) est le premier algorithme avec correction d'erreur locale dont nous parlerons dans cette section. Il s'agit d'une technique basée sur l'algorithme d'Earley où la correction locale comportant le moins d'erreurs est choisie. L'idée générale est de transformer une phrase contenant des erreurs en une phrase n'en contenant pas tout en respectant la grammaire hors contexte. L'algorithme d'Earley est alors exécuté sur la phrase corrigée, ce qui a pour effet que l'analyse va toujours être un succès puisque la phrase corrigée respecte la grammaire. Pour parvenir à corriger la phrase originale, l'algorithme se base sur les notions de coût et de programmation dynamique et s'inspire grandement de la solution au problème de correction mot vers mot (*The String-to-String Correction Problem*, (Wagner et Fischer, 1974)).

L'algorithme associe un coût aux différentes hypothèses d'erreurs : insertion, mutation et suppression d'un terminal. Cette définition est ensuite étendue aux séquences de terminaux. Ensuite, pour chaque mot de la phrase, l'algorithme détermine la meilleure opération à effectuer. Comme cet algorithme a été choisi en vue d'une évaluation de ses performances, une description détaillée est donnée à la section 3.1.

Nous pouvons égaler mentionner les articles suivants qui présentent d'autres algorithmes de correction locale des erreurs : (Irons, 1963) et (Dion et Fischer, 1978).

2.2.2 Correction d'erreur régionale

Les algorithmes de correction d'erreurs sont un compromis entre la correction locale et la correction globale. C'est une manière d'avoir le meilleur des deux mondes : des corrections de très grande qualité tout en gardant une bonne vitesse d'exécution. Leur habileté à activer la correction seulement pour les régions comportant des erreurs est ce qui les rend plus rapides que les algorithmes de correction globale.

Les algorithmes de correction d'erreur régionale mentionnés ci-dessous travaillent tous de la même façon. Tant qu'une erreur n'est pas rencontrée, l'algorithme travaille en mode sans correction d'erreur. Quand l'algorithme détecte une erreur, on parle alors du mot t_i comme étant le point de détection de l'erreur. L'algorithme tente alors de déterminer le point d'erreur qui peut se trouver avant le mot t_i dans la phrase. Puis, l'algorithme détermine la longueur totale de la région de correction à évaluer. Quand la région est obtenue, l'algorithme active la correction d'erreur seulement pour le contexte de la région et non pour la phrase entière.

Les algorithmes se distinguent entre eux de deux manières. La première est la façon dont la région est choisie, c'est-à-dire combien de mouvements arrière et combien de mouvements avant sont requis pour déterminer la région. La deuxième est le moyen qu'ils utilisent pour effectuer la correction d'erreur pour la région en question. Seulement quelques algorithmes de correction régionale ont été développés au cours des dernières années. L'attrait est cependant de plus en plus grand car ils sont très prometteurs.

Le premier article élaborant une technique de correction d'erreur régionale (Lévy, 1975) a été publié en 1975. L'algorithme provient de la thèse de doctorat de l'auteur. L'article définit en premier lieu la notion de *beacon*. Les *beacon* sont les terminaux de la grammaire qui font en sorte que tout se qui se trouve avant eux forme des préfixes valides du langage généré par la grammaire. Ces *beacons* peuvent être déterminés avant de commencer l'analyse de la phrase. Le retour en arrière consiste simplement à retourner au dernier *beacon* rencontré. Pour le mouvement avant, qui détermine le contexte à droite du point d'erreur, l'auteur utilise une méthode de génération. À partir du point d'erreur, on génère toutes les sous-phrases pouvant se trouver après le point d'erreur. Quand toutes ces interprétations sont équivalentes, on arrête la génération. Les interprétations sont dites équivalentes lorsqu'un suffixe d'une sous-phrase peut être aussi un suffixe de toutes les autres sous-phrases générées. À ce point, on a une région ainsi qu'un ensemble de sous-phrases pouvant s'insérer dans cette région. L'auteur choisit alors la séquence de mots générée requérant le moins d'opérations d'édition pour être équivalente à la séquence délimitée par la région. L'algorithme continue alors son processus normal avec ce qui suit la sous-phrase choisie et recommence le même stratagème s'il rencontre d'autres erreurs.

Plus récemment, une équipe provenant d'Espagne a proposé un nouvel algorithme de correction régionale des erreurs. L'article en question (Vilares *et al.*, 2000) définit une nouvelle approche pour rendre un algorithme d'analyse syntaxique classique robuste aux erreurs. L'idée peut s'appliquer à n'importe quel algorithme dit de décalage-réduction (ou *shift-reduce* en anglais) tel que les classes d'algorithme LR ou *Look-Ahead LR parser* (LALR). Il est également aisé selon les auteurs de transposer l'idée à des algorithmes plus généraux. Lorsque l'analyseur rencontrent une erreur, il définit à cet endroit un point d'erreur. Une liste d'items

de détection est alors créée. Ces items de détection se trouvent avant ou directement au point d'erreur. Un item de détection correspond en quelque sorte à un mouvement arrière dans la chaîne à analyser. La particularité ici est qu'il y a plusieurs items de détection et donc, plusieurs mouvements arrière évalués indépendament les uns des autres. Les items de détection possèdent un compteur d'erreur pour garder une trace du nombre d'erreurs. Ensuite, pour chaque items de détection, l'algorithme recommence l'analyse mais cette fois en augmentant l'ensemble des transitions possibles avec des transitions d'erreurs. À chaque fois qu'une transition d'erreur est appliquée, le compteur d'erreur de l'item de déctection est incrémenté. On continue d'appliquer des transitions jusqu'à ce que l'analyseur soit en mesure de reprendre son cycle normal d'opérations. À ce point, une liste de transitions pour chaque item de détection est obtenue. Il ne reste plus qu'à déterminer la liste de transitions ayant générée le moins d'erreur. Cette liste est choisie et appliquée à l'analyseur. Le processus est ensuite répété jusqu'à ce que toute la chaîne ait été analysée.

Mentionnons également les articles (Tai, 1978) et (Mauney et Fischer, 1982) qui développent également des algorithmes de correction régionale des erreurs.

2.2.3 Correction d'erreur globale

La correction globale d'erreurs est la catégorie des algorithmes trouvant toujours la correction contenant le moins d'erreurs pour la phrase dans sa globalité. Ce faisant, les algorithmes contenus dans cette catégorie donnent de meilleures corrections que les algorithmes effectuant une correction locale ou régionale des erreurs. Cette qualité dans les corrections proposées a bien sûr un coût : un temps de calcul plus élevé. En effet, pour offrir la correction contenant le moins d'erreurs, les algorithmes doivent explorer une bien plus grande quantité d'hypothèses que les autres classes d'algorithmes d'analyse syntaxique robustes.

Cette classe d'algorithmes est probablement la catégorie ayant reçu le plus d'attention des chercheurs au cours des trois dernières décennies. La recherche s'est particulièrement attelée à la tâche de diminuer le temps écoulé lors de l'analyse. La raison derrière ceci est que les premiers algorithmes de ce genre n'étaient pas très utilisables pour effectuer l'analyse de phrases comportant beaucoup de mots. Encore aujourd'hui, peu d'avancements ont été effectués au niveau du temps de calcul des algorithmes offrant une correction globale. Une des avenues les plus explorées pour réduire le temps de calcul est de limiter l'espace de recherche des solutions possibles. Malgré cela, il est encore difficile de réduire l'espace tout en laissant intacte la qualité des corrections proposées par ces algorithmes. Dans le contexte précis du projet, les entrées sont relativement petites, une cinquantaine de mots pour une phrase. L'utilisation de tels algorithmes est donc envisageable. En comparaison, l'analyse syntaxique d'un programme informatique complet doit s'effectuer bien souvent sur plus de

1000 terminaux, ce qui vient mettre à genoux les algorithmes de correction globale.

Une des premiers articles traitant de correction globale des erreurs a été publié en 1972. Cet article (Aho et Peterson, 1972) décrit une manière de faire de la correction d'erreurs en modifiant la grammaire. L'idée est d'ajouter des productions à la grammaire, appelées productions d'erreurs, de tel sorte que la grammaire augmentée couvre la grammaire originale tout en générant l'ensemble T^*. Cet ensemble représente toutes les phrases pouvant être générée en utilisant les terminaux contenus dans l'ensemble T.

Après avoir étendu la grammaire, les auteurs utilisent un algorithme d'Earley légèrement modifié pour faire l'analyse des phrases. Les items d'Earley sont augmentés d'un compteur d'erreur. L'algorithme procède comme la version originale, sauf quand vient le temps de la phase de complétion. À cette phase, un item peut être débloqué par un item soit lié à une production normale, soit lié à une production d'erreur. Dans les deux cas, le compteur d'erreur de l'item bloqué (e_b) est incrémenté du compteur d'erreur de l'item qui l'a débloqué. Cependant, dans le deuxième cas, on incrémente en plus e_b du nombre d'erreurs contenu dans la production d'erreur.

Cette additivité des erreurs fait en sorte qu'un item bloqué contenant 2 erreurs et débloqué par un item qui en contient 3 lié à une production d'erreur de 1 erreur résultera en un item débloqué contenant 6 erreurs ($2 + 3 + 1$). Quand un item bloqué vient d'être avancé, il est ajouté à l'ensemble d'Earley courant seulement s'il ne contient pas déjà un item identique, mais ayant un nombre d'erreurs plus petit. Dans le cas où un item semblable existe, mais avec un compteur d'erreurs plus grand, ce dernier est remplacé par le nouvel item contenant moins d'erreurs. Ces conditions permettent d'assurer l'optimalité des corrections.

Un des articles les plus cités (Lyon, 1974) reprend essentiellement l'idée de base de l'algorithme précédent, mais cette fois, sans faire appel à des productions d'erreurs. Cet algorithme, développé par Gordon Lyon, a également comme base l'algorithme d'Earley. Pour parvenir à le rendre robuste aux erreurs, l'auteur introduit également un compteur d'erreur aux items d'Earley. Il s'assure lui aussi que les items présents dans un ensemble d'Earley ont toujours un compteur d'erreur minimal. L'idée principale est de tenir compte de certaines hypothèses d'erreur lors de la phase de scan. Dans cette phase, quatre hypothèses sont envisagées : un match parfait avec le terminal courant, une mutation, une suppression ou une insertion d'un terminal. Pour les trois dernières, quand le *dotOffset* de l'item est avancé, le compteur d'erreur est incrémenté de 1. Pour la phase de complétion, l'algorithme fonctionne comme l'algorithme d'Aho et Peterson. L'algorithme de Lyon sera décrit ultérieurement dans la section 3.2.

L'algorithme précédent ne prend pas en considération tous les cas de figure, par exemple lorsque deux non-terminaux sont interchangés. Un algorithme (Lee *et al.*, 1995) a été développé étendant la technique de Lyon avec d'autres hypothèses d'erreurs. Au lieu de travailler ex-

clusivement avec des hypothèses d'erreurs pour les terminaux, ce nouvel algorithme travaille également sur les non-terminaux. Les hypothèses d'erreurs ajoutées sont : la suppression, l'insertion et la mutation d'un non-terminal. Ces hypothèses supplémentaires sont traitées dans la phase de complétion de l'algorithme de Lyon. Cette technique utilise différentes heuristiques, certaines définies dans l'article original de Lyon, afin de réduire l'espace de recherche et ainsi obtenir des performances plus acceptables.

Un article publié en 1993 (Kuroda et Tanaka, 1993) reprend également l'idée de Lyon et l'applique à l'analyseur syntaxique de Leiss. La technique décrite dans cet article est semblable à celle utilisée par Lyon, mais adaptée à un autre algorithme d'analyse syntaxique. Cet autre algorithme (Leiss, 1990) est en fait une version améliorée de l'algorithme original d'Earley. Dans cette version améliorée, beaucoup moins d'items sont générés comparativement à l'algorithme original. C'est donc dire que la version avec correction d'erreurs produit également moins d'items. Les auteurs affirment que leur algorithme garde la même qualité au niveau de la correction des erreurs que l'algorithme de Lyon tout en étant plus rapide.

Un autre article traitant de la correction d'erreurs globale (Pighizzini, 1992) tente plutôt d'améliorer le temps de calcul en parallélisant le processus d'analyse syntaxique. Dans cet article, l'auteur étend un algorithme parallèle sans correction (Bruschi et Pighizzini, 1992) afin de le rendre robuste aux erreurs. Pour y arriver, l'article introduit le concept d'arbre de reconnaissance, ou *recognition tree* en anglais. Cet arbre est un arbre binaire décomposant un item, identique aux items d'Earley, en leurs sous-composantes. Pour un noeud associé à un item de la forme $[A \rightarrow \alpha\ B\ \bullet, 0, 6]$ et une phrase reconnue **a b c d e f**, on assume que α a reconnu **a b c** et que le non-terminal B a reconnu **d e f**. Alors l'enfant de droite aura comme valeur $[B \rightarrow \beta\ \bullet, 3, 6]$ et l'enfant de gauche la valeur $[A \rightarrow \alpha \bullet B, 0, 3]$. Les indices 3 et 6 de l'enfant de droite proviennent du fait que cet enfant a reconnu **d e f** qui commence à l'ensemble 3 et se termine à l'ensemble 6. Le même raisonnement s'applique pour le noeud de gauche. Ces noeuds enfants sont également des arbres de reconnaissances. Chaque noeud est ensuite augmenté d'un poids qui représente le nombre d'erreurs à ce noeud. Ceci permet alors de choisir le chemin ayant le moins d'erreurs. Différents chemins sont alors évalués en parallèle, ce qui permet un gain de performance sur des machines contenant plusieurs processeurs.

CHAPITRE 3

ANALYSE SYNTAXIQUE ROBUSTE

Ce chapitre porte sur les techniques d'analyse syntaxique robuste qui ont été évaluées dans le cadre de ce projet de maîtrise. Il s'attarde principalement à la description des trois algorithmes évalués ainsi que leurs implémentations.

La première section porte sur l'algorithme d'Anderson-Backhouse, un algorithme de correction locale des erreurs. La deuxième partie porte quant à elle sur l'algorithme de Lyon qui effectue plutôt une correction globale des erreurs. Le chapitre se termine avec une description de notre algorithme hybride qui est une fusion entre l'algorithme classique d'Earley et l'algorithme de Lyon.

3.1 Algorithme d'Anderson-Backhouse

L'algorithme dont il est question dans cette section a été développé en 1979 et présenté deux ans plus tard dans le journal « *ACM Transactions on Programming Languages and Systems* » sous le nom « *Locally Least-Cost Error Recovery in Earley's Algorithm* ». Voir (Anderson et Backhouse, 1981) pour l'article complet.

Cet algorithme permet de faire de la correction d'erreurs d'une manière locale en utilisant l'algorithme d'Earley comme base. Les auteurs ont développé l'algorithme avec deux buts bien précis en tête : choisir la correction ayant un coût minimal parmi toutes les corrections possibles et ne pas engendrer de nouvelles erreurs de syntaxe lors du choix d'une correction particulière. L'algorithmique derrière la technique est dérivée de la programmation dynamique.

L'algorithme reçoit en entrée une grammaire CFG $G = (N, T, P, S)$ ainsi qu'une liste de mots $t_0, t_1, \ldots, t_{n-2}, t_{n-1}$ où n est le nombre de mots dans la liste. L'élément t_{n-1}, le dernier de la liste, doit toujours être le symbole $\dashv$. Ce symbole représente la fin d'une phrase et est considéré comme un terminal, il fait donc partie de l'ensemble T. S'il n'est pas présent à la fin de la phrase reçue en entrée, il est ajouté automatiquement.

Le but de l'algorithme d'Anderson-Backhouse est de produire, à partir de la phrase originale, une phrase corrigée à coût minimal qui respecte la grammaire G. S'il réussit à satisfaire ces critères, il aura fourni une version corrigée valide de la phrase et un algorithme classique sera en mesure de l'analyser correctement.

Pour produire cette phrase corrigée, l'algorithme parcourt la phrase originale mot à mot

et produit une seconde liste de mots $u_0, u_1, \ldots, u_{n_c-2}, u_{n_c-1}$ où n_c est le nombre de mots corrigés. Cette seconde liste de mots représente la phrase corrigée. Pour chaque mot de la phrase originale, l'algorithme détermine la meilleure correction à appliquer. En fonction de la décision prise, la liste de mots corrigée sera ajustée en conséquence. Il est bon de noter que la phrase corrigée ne contiendra pas nécessairement le même nombre de mots que la phrase originale.

L'algorithme effectue son travail sans jamais reconsidérer ses décisions passées. Chaque décision prise est la meilleure localement, mais pas forcément globalement parce l'algorithme ne considère pas toutes les corrections théoriquement envisageables. Ce compromis permet d'obtenir un algorithme plus rapide qu'un algorithme de correction globale tout en offrant des corrections de haute qualité.

Au tout début de l'algorithme d'Anderson-Backhouse, une première étape d'initialisation est effectuée. Durant cette étape, l'algorithme initialise les différentes variables qui seront accessibles aux fonctions qui le compose. L'algorithme 3.1 décrit plus en détails cette fonction.

Algorithme 3.1 Fonction *initialize* de l'algorithme d'Anderson-Backhouse

Entrée(s) : Grammaire $G = (N, T, P, S)$ et liste de mots $t_0, t_1, \ldots, t_{n-1}$

$i \leftarrow 0$
$j \leftarrow 0$
$u \leftarrow [\,]$

$P \leftarrow P \cup \{Z \rightarrow S \dashv\}$

$s_0 \leftarrow \text{EarleySet}()$
$s_0 \leftarrow s_0 \cup \{[Z \rightarrow \bullet\ S \dashv, 0, 0]\}$

predict(s_0)
complete(s_0)

La variable i stocke en mémoire l'index du mot original présentement analysé et la variable j fait le même travail mais pour la liste de mots corrigés. La variable u est une liste et est initialisée avec la valeur $[\,]$ qui représente la liste vide.

On peut voir que la fonction *initialize* fait appel aux fonctions *predict* et *complete*. La fonction *complete* est identique à la fonction originale d'Earley. La fonction *predict* est quant à elle fortement inspirée de l'algorithme d'Earley. Elle effectue le même travail mais est également chargée d'une étape supplémentaire. Cette étape consiste à calculer le coût de complétion des items de prédiction de l'ensemble d'Earley s_j. Les items de prédictions sont les items bloqués à un non-terminal, c'est-à-dire de la forme $[A \rightarrow \alpha \bullet B\,\beta, f, j]$. L'algorithme 3.2 dé-

crit la fonction *predict* modifiée de l'algorithme d'Anderson-Backhouse. L'appel à la fonction *predictCompletionCosts* est l'étape supplémentaire qui calcule les coûts de complétion. Les détails de cette fonction sont donnés à l'annexe A par l'algorithme A.1.

Algorithme 3.2 Fonction *predict* de l'algorithme d'Anderson-Backhouse

predictCompletionCosts(s_j)

earleyPredict(s_j)

Après la phase d'initialisation, l'algorithme entre dans sa boucle principale. Cette dernière continue tant que le dernier élément de la liste de mots corrigés n'est pas équivalent à $\dashv$ ($u_j \neq \dashv$). Pendant cette boucle, l'algorithme traversera mot à mot la phrase originale. Le mot original présentement analysé est donc accessible grâce à la variable t_i.

À chaque itération de la boucle, l'algorithme détermine la meilleure action à appliquer à la phrase corrigée en fonction du mot t_i. L'algorithme peut choisir cinq opérations d'édition distinctes pour le mot t_i.

1. Il peut décider d'accepter le mot t_i comme étant valide et l'ajouter à la position j dans la phrase corrigée.
2. Il peut décider de supprimer t_i et donc ne rien n'ajouter de nouveau dans la phrase corrigée car le terminal ne devrait pas se retrouver dans la phrase corrigée.
3. Il peut décider qu'un nouveau mot devrait être placé devant t_i et ajoute ce nouveau mot à la position j dans la phrase corrigée.
4. Il peut décider de changer le mot t_i par un autre qui provient de l'ensemble T. Ce nouveau mot, qui remplace t_i, est ajouté à la position j dans la phrase corrigée.
5. Il peut décider de ne rien faire. Ceci implique qu'il n'y a aucune correction valable pour le mot t_i.

Chaque opération d'édition est encodée dans une structure de données appelée *EditOperation*. Cette structure contient quatre champs différents. Le premier, nommé *type* contient le type de l'opération d'édition. Les types peuvent être *Accept*, *Delete*, *Insert*, *Change* et *None*. Le type *None* est spécial puisqu'il représente une opération ne pouvant pas être appliquée à la phrase originale. Cette opération est utilisée pour relayer le fait qu'aucune correction n'a pu être trouvée.

La structure contient également les champs *terminal* et *oldTerminal*. Ces deux champs stockent respectivement le nouveau terminal obtenu après l'opération d'édition et l'ancien terminal qui était présent. Le tableau 3.1 énumère les différentes valeurs de ces deux variables

en fonction de l'opération d'édition utilisée. Finalement, la structure *EditOperation* contient également un champ nommé *errorCount* qui contient un compteur d'erreurs. Ce compteur d'erreurs correspond au coût de l'opération d'édition qui a été appliquée.

Tableau 3.1 Correspondance entre l'opération d'édition et les valeurs de *terminal* et *oldTerminal*

Opération	*Terminal*	*oldTerminal*
Accepter t	t	t
Supprimer t	ϵ	t
Insérer t'	t'	ϵ
Changer t par t'	t'	t
Aucune opération	ϵ	ϵ

L'opération d'édition est choisie par la fonction *findBestEditOperation*, qui évalue différentes hypothèses afin d'arrêter son choix. Si c'est l'opération de type *None* qui est retournée, l'algorithme s'arrête immédiatement, car cela indique qu'il n'y a pas de correction possible.

Lorsque l'opération retournée est de supprimer le mot t_i, l'algorithme effectue un traitement particulier. Dans ce cas, l'algorithme n'ajoute rien dans la phrase corrigée car le mot t_i est à supprimer. L'algorithme incrémente ensuite i de 1 et détermine une nouvelle opération d'édition pour le nouveau mot t_i. Ce processus est répété dans une boucle tant que l'opération d'édition choisie est une opération de suppression.

Ceci implique que lorsque la boucle se termine, l'algorithme est assuré que l'opération d'édition choisie va permettre d'ajouter un mot à la phrase corrigée. Ce mot est ajouté à la fin de la liste u. La variable j est alors incrémentée de 1 pour refléter cette addition. La variable i est également incrémentée de 1 si l'opération d'édition n'était pas une opération d'insertion. En effet, si l'opération est d'insérer un nouveau mot avant t_i, ceci implique que t_i n'a pas encore été traité et que l'algorithme doit choisir une autre opération pour le mot t_i.

Après avoir ajouté un mot corrigé dans la liste u et incrémenté les compteurs, l'algorithme effectue les phases de scan, de prédiction et de complétion de l'algorithme d'Earley classique sur le mot corrigé. Ces phases permettront de peupler les ensembles d'Earley s_j et s_{j+1}. Il faut bien comprendre ici que l'algorithme d'Earley classique effectue son travail sur la phrase corrigée. C'est donc la liste u de mots corrigés qui est utilisée par l'algorithme et non la liste t.

Ce processus est répété jusqu'à ce qu'on ait parcouru la liste de mots originaux au complet. Lorsque l'algorithme a terminé de parcourir la phrase originale, les mots $u_0, u_1, \ldots, u_{n_c-2}, u_{n_c-1}$ forment la version corrigée de la phrase entrée par l'utilisateur.

L'algorithme 3.3 décrit la fonction *recognize* qui s'occupe de faire la reconnaissance d'une phrase. Les fonctions *scan* et *complete* sont identiques aux fonctions originales d'Earley tandis que la fonction *predict* a été définie plus haut par l'algorithme 3.2.

Algorithme 3.3 Fonction *recognize* de l'algorithme d'Anderson-Backhouse

```
initialize()

tant que u_j ≠ ⊣ faire
    editOperation ← findBestEditOperation()
    si editOperation.type est None alors
        exit(« Aucune correction possible »)

    tant que editOperation.type est Delete faire
        i ← i + 1
        editOperation ← findBestEditOperation()

    u ← u + [editOperation.terminal]
    j ← j + 1
    si editOperation.type n'est pas Insert alors
        i ← i + 1

    scan(s_j, s_{j+1})
    predict(s_j)
    complete(s_j)
```

Quand on regarde l'algorithme 3.3, on voit rapidement que la partie centrale de l'algorithme se trouve dans la sélection de la prochaine meilleure opération d'édition. Cette sélection s'effectue via la fonction *findBestEditOperation*.

L'algorithme 3.4 illustre le processus effectué par la fonction *findBestEditOperation*. Tout d'abord, l'algorithme débute en initialisant la variable *best* qui stockera la meilleure opération d'édition trouvée. Au départ, c'est l'opération d'édition de suppression du terminal courant t_i qui est sélectionnée par défaut. L'algorithme boucle ensuite sur tous les items bloqués à un terminal dans l'ensemble s_j, c'est-à-dire de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, j]$. À chaque itération de la boucle, la fonction *findBestEditOperation* évaluara deux hypothèses.

Elle regarde en premier si le coût d'incomplétion pour cet item est inférieur au coût de l'opération présentement stockée dans la variable *best*. Si oui, alors il utilise le vecteur *incompleteCostOperation* pour déterminer la meilleure opération possible et remplace *best* par cette dernière. Le coût d'incomplétion est le coût minimal pour placer t_i dans l'item sans toutefois compléter l'item en question.

La fonction calcule en deuxième le coût d'insérer $\mathbf{t}\ \beta$ (donné par $cost^*(\epsilon, \mathbf{t}\ \beta)$) aditionné au coût de compléter ce qui suit A (donné par $completionCosts[f][A, t_i]$). La fonction $cost^*$ calcul le coût d'insérer une séquence de symboles tandis que l'élément *completionCosts* est le vecteur des coûts de complétion.

Quand le coût calculé est inférieur au coût de l'opération stockée dans *best*, la nouvelle opération d'édition devient l'opération d'insérer $\mathbf{t}$. Ces deux opérations sont répétées pour tous les items de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, j]$ dans l'ensemble s_j. À la fin de la boucle, la meilleure opération d'édition trouvée est retournée.

Algorithme 3.4 Fonction *findBestEditOperation* de l'algorithme d'Anderson-Backhouse

Entrée(s) : s_j l'ensemble d'Earley présentement analysé
  $best \leftarrow EditOperation(Delete, t_i, \epsilon, cost^*(t_i, \epsilon))$
  pour tout *item* **de la forme** $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, j]$ **dans** s_j **faire**
    si $incompleteCost^*(t_i, prefixes(\mathbf{t}\ \beta)) < best.cost$ **alors**
      $best \leftarrow incompleteCostOperation(t_i, \mathbf{t}\ \beta)$

    $cost \leftarrow cost^*(\epsilon, \mathbf{t}\ \beta) + completionCosts[f][A, t_i]$
    si $cost < best.cost$ **alors**
      $best \leftarrow EditOperation(Insert, \epsilon, \mathbf{t}, cost)$
  retourner *best*

Dans les paragraphes qui suivent, nous allons nous attarder à expliquer en détails les différentes fonctions utilisées par l'algorithme 3.4. Nous débuterons par la fonction *cost* qui est la base des fonctions $cost^*$ et $incompleteCost^*$. Les détails pour calculer le vecteur *completionCosts* ne sont pas abordés dans ce chapitre. Pour une description détaillée du vecteur des coûts de complétion, se référer à l'annexe A.

L'inspiration des auteurs pour la fonction *cost* provient de l'algorithme de correction d'erreur dans des chaînes de caractères (Wagner et Fischer, 1974). Cette technique, basée sur la programmation dynamique, a ensuite été adaptée par Anderson et Backhouse aux grammaires CFG.

L'idée est de commencer par la correction d'un terminal, ce que nous appellons une opération d'édition. Chaque opération est associée à un coût, ce coût peut être fixe ou dynamique en fonction de l'opération d'édition et/ou du terminal présentement analysé. Le coût peut également être infini, dénotant alors que l'opération en question ne peut pas être appliquée. Ceci est utile pour limiter les corrections possibles. Le choix de la meilleure opération s'effectue en choisissant l'opération ayant le coût le plus faible, si elle existe. Il suffit ensuite d'étendre cette idée par induction à la chaîne au complet.

Dans le cadre strict des grammaires hors contexte, il y a certaines contraintes au niveau des coûts qui doivent être respectées. Par exemple, il n'est pas valide de supprimer, de changer ou d'insérer le symbole $\dashv$ qui représente la fin de la phrase. Cette contrainte doit être respectée car dans le cas contraire, l'algorithme n'aurait pas de moyen sûr de déterminer la fin de l'analyse. Cela pourrait engendrer la fâcheuse conséquence qu'il serait alors possible de supprimer le symbole de fin de phrase puis d'ajouter une série infinie de terminaux.

L'équation 3.1 définit la fonction utilisée pour déterminer le coût d'une opération d'édition sur un terminal. Nommée $cost(left, right)$, elle retourne le coût pour changer un symbole ($left$) en un autre ($right$). Le domaine des variables reçues en entrée est $T \cup \{\epsilon\}$ tandis que la plage des valeurs de sortie est compris entre 0 et ∞.

$$cost(left, right) = \begin{cases} 0 & \text{si } left = right \text{ (Entrées identiques)} \\ \infty & \text{si } left = \dashv \vee right = \dashv \text{ (Modification du symbole } \dashv) \\ x & \text{si } left = \epsilon \wedge right \in T \text{ (Insertion d'un terminal)} \\ y & \text{si } left \in T \wedge right = \epsilon \text{ (Suppression d'un terminal)} \\ z & \text{si } left \in T \wedge right \in T \wedge left \neq right \text{ (Mutation d'un terminal)} \end{cases} \tag{3.1}$$

Le deuxième cas de l'équation modélise la contrainte que le symbole $\dashv$ ne peut être ni inséré, ni supprimé, ni muté. La seule opération est l'acceptation du symbole $\dashv$. Les variables x, y et z de l'équation 3.1 sont choisies en fonction des besoins spécifiques de l'application qui utilise l'algorithme et doivent être strictement positives.

Cependant, cette fonction travaille exclusivement sur un seul terminal à la fois, pour nous être utile, il nous faut étendre cette définition de manière à pouvoir travailler sur une séquence de terminaux. Nous nommons cette fonction $cost*$ et sa déclaration est $cost*(t, v)$ où $t \in T \cup \{\epsilon\}$ et $v \in T^*$.

Il y a deux cas de figure pour la fonction $cost*$: le cas où $t = \epsilon$ et le cas où $t \in T$. Le premier cas correspond à l'insertion d'une séquence de terminaux, car on cherche à obtenir le coût pour remplacer la chaîne vide par une séquence de terminaux. Dans le deuxième cas, on cherche plutôt le coût de transformer le terminal en la séquence v. Dans les deux cas, c'est le coût de transformation minimal qu'on cherche à obtenir.

La définition 3.2 représente la définition de $cost*$ dans le cas d'une insertion de terminaux. Elle dit simplement que le coût d'insérer une séquence de terminaux est équivalent au coût d'insérer une séquence composée des $m-1$ premiers éléments plus le coût d'insérer le dernier terminal de la séquence. La variable m correspond au nombre de terminaux dans la séquence v.

$$cost^*(\epsilon, v) = \begin{cases} 0 & \text{si } v = \epsilon \\ cost^*(\epsilon, w) + cost(\epsilon, t') & \text{si } v = wt' \mid w \in T^* \wedge t' \in T \end{cases} \tag{3.2}$$

Lorsque la fonction $cost^*(t, v)$ reçoit un terminal plutôt que le symbole ϵ comme premier argument, la description est un petit peu plus complexe. Dans ce cas, il faut évaluer toutes les combinaisons possibles pour changer le terminal en une séquence. Le cas de base survient quand $v = \epsilon$ ce qui correspond à supprimer le terminal t. Le coût pour supprimer t est obtenu en faisant appel à la fonction *cost* de cette manière : $cost(t, \epsilon)$.

Les autre cas surviennent quand v est de la forme $v = wt' \mid w \in T^* \wedge t' \in T$. On utilise alors le coût résultant minimal parmi les trois possibilités de décompositions qui suivent.

1. Le coût d'insérer v plus le coût de supprimer t.
2. Le coût d'insérer w plus le coût de changer t en t'.
3. Le coût de changer t par la séquence de terminaux w plus le coût d'insérer t'.

La définition 3.3 donne une définition formelle de la fonction $cost^*$ lorsque $t \neq \epsilon$.

$$cost^*(t, v) = \begin{cases} cost(t, \epsilon) & \text{si } v = \epsilon \\ \min \begin{pmatrix} cost^*(\epsilon, v) + cost(t, \epsilon), \\ cost^*(\epsilon, w) + cost(t, t'), \\ cost^*(t, w) + cost(\epsilon, t') \end{pmatrix} & \text{si } v = wt' \mid w \in T^*, \ t \text{ et } t' \in T \end{cases} \tag{3.3}$$

Pour que l'algorithme puisse utiliser adéquatement et facilement la fonction $cost^*(t, v)$, cette dernière doit être en mesure de traiter non seulement des séquences de terminaux, mais également des séquences composées à la fois de terminaux et de non-terminaux. Par conséquent, elle doit pouvoir traiter v tel que $v \in (N \cup T)^*$. Les définitions 3.2 et 3.3 doivent être étendues pour répondre à cette exigence. La technique est très simple, il suffit de générer les chaînes pouvant être produites par les non-terminaux et d'obtenir la correction ayant le coût minimal en fonction des chaînes générées.

La fonction $L(\alpha)$ reçoit une séquence de symboles, c'est-à-dire que $\alpha \in (N \cup T)^*$, et génère toutes les séquences de terminaux pouvant être dérivées de la grammaire G. Cette fonction est donnée à la définition 3.4.

$$L(\alpha) = \{v \mid v \in T^* \wedge \alpha \underset{G}{\Rightarrow}^* v\} \tag{3.4}$$

Par exemple, si on a la séquence **a** A **c** et que la production A correspond à $A \rightarrow$ **b1** | **b2** | **b3**, alors on peut générer trois chaînes : **a b1 c**, **a b2 c** et **a b3 c**. Il suffit ensuite

de parcourir les chaînes générées, d'appeler *cost**(t, v) pour chacune d'entre elles et de ne garder que le coût minimal. La définition 3.5 exprime plus formellement cette description.

$$\textit{cost*}(t, \alpha) = \min(\{\textit{cost*}(t, v) \mid \forall v \in L(\alpha)\}) \tag{3.5}$$

La description du concept de coût ayant été donnée, nous nous attardons dès à présent sur la fonction qui calcule le coût d'incomplétion. Cette dernière, déclarée comme *incompletionCost**(t, v) où $t \in T$ et $v \in T^+$ utilise la fonction *cost** dans sa définition.

Pour pouvoir bien décrire *incompletionCost**(t, v), nous devons tout d'abord introduire la fonction *prefixes* (3.6). Cette dernière est en effet utilisée en tandem avec la fonction *incompletionCost**, lorsque qu'appelée dans la fonction *findBestEditOperation* (3.4) : *incompleteCost**$(t_i, \textit{prefixes}(\mathbf{t}\ \beta))$.

La fonction *prefixes* permet d'obtenir la liste de tous les préfixes non vides pouvant être dérivés de la séquence $\mathbf{t}\ \beta$. Par exemple, si $\mathbf{t}\ \beta$ équivaut à la chaîne **a b c**, alors les préfixes non vides sont **a**, **a b** et **a b c**.

$$\textit{prefixes}(\alpha) = \left\{v \mid \forall vw = \alpha \wedge v \in T^+ \wedge w \in T^*\right\} \tag{3.6}$$

La fonction *incompletionCost** semble effectuer le même travail que *cost** mais il y une différence importante. En effet, la fonction *incompletionCost** ne s'occupe que d'un seul cas, celui d'insérer une séquence de terminaux et de remplacer t par le dernier symbole de cette séquence. Par exemple, le résultat de *incompleteCost**$(\mathbf{c}, [\mathbf{a}, \mathbf{b}, \mathbf{d}])$ sera donné par *cost**$(\epsilon, [\mathbf{a}, \mathbf{b}]) + \textit{cost}(\mathbf{c}, \mathbf{d})$.

Dans le cas où $v = \epsilon$, la fonction retourne la valeur infinie. Ceci empêche de remplacer t par ϵ, ce qui signifirait de supprimer t. Dans le cas contraire, la fonction calcule simplement le coût d'insérer les $m - 1$ premiers caractères de v additionnés au coût de remplacer t par un des terminaux se trouvant dans l'ensemble T.

$$\textit{incompleteCost*}(t, v) = \begin{cases} \infty & \text{si } v = \epsilon \\ \textit{cost*}(\epsilon, w) + \textit{cost}(t, t') & \text{si } v = wt' \mid w \in T^* \wedge t' \in T \end{cases} \tag{3.7}$$

Tout comme la fonction *cost**, la définition 3.7 doit également être étendue de manière à pouvoir traiter un argument v contenant une séquence de terminaux et non-terminaux. Cette extension s'effectue de manière similaire à ce qui a été fait pour la définition 3.5.

On se rappelle que dans la fonction *findBestEditOperation*, la fonction *incompletionCost** est utilisée comme suit : *incompleteCost**$(t_i, \textit{prefixes}(\mathbf{t}\ \beta))$. La variable t_i est le mot original présentement analysé et $\mathbf{t}\ \beta$ provient de l'item $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, c]$. Puisque la fonction

prefixes retourne un ensemble, la fonction *incompleteCost** doit être adaptée afin de traiter un ensemble. Donc, quand la fonction *incompleteCost** reçoit un ensemble comme deuxième argument plutôt qu'une liste de symboles, elle effectue un traitement différent. Elle itère sur chaque élément de l'ensemble et appele la fonction originale avec cet élément. Elle choisit ensuite l'élément ayant retourné le coût le plus petit.

L'utilisation de la fonction *prefixes* lors de l'appel à *incompletionCost** est primordiale, car elle permet d'obtenir le coût d'incomplétion des sous-séquences de **t** β. L'utilisation des préfixes permet à l'algorithme de traiter les cas suivants pour le terminal t_i : accepter t_i, changer t_i par **t** ou insérer un ou plusieurs terminaux provenant de **t** β puis changer t_i par le symbole suivant la séquence insérée.

Pour bien voir les cas englobés par l'utilisation de la fonction *prefixes*, voici quelques exemples pour chacun d'eux ainsi que l'opération d'édition associée.

1. Accepter t_i
   L'algorithme analyse présentement t_i qui équivaut à **b** et faisant partie de la chaîne initiale non corrigée **a b c**. L'item analysé est $[T \rightarrow \mathbf{a} \bullet \mathbf{b}\ \mathbf{c}, f, c]$. Le premier préfixe est **b**. Le coût est obtenu avec *incompleteCost**$(\mathbf{b}, \mathbf{b})$ et équivaut à zéro (le coût d'insérer zéro terminal plus le coût de changer **b** par **b**). L'opération d'édition pour ce coût est donc d'accepter le terminal t_i.
2. Changer t_i
   L'algorithme analyse présentement t_i qui équivaut à **d** et faisant partie de la chaîne initiale non corrigée **a d c**. L'item analysé est $[T \rightarrow \mathbf{a} \bullet \mathbf{b}\ \mathbf{c}, f, c]$. Le premier préfixe est **b**. Le coût est obtenu avec *incompleteCost**$(\mathbf{d}, \mathbf{b})$ et équivaut au coût de changer **d** par **b** (le coût d'insérer zéro terminal plus le coût de changer **d** par **b**). L'opération d'édition pour ce coût est donc de changer le terminal t_i par **b**.
3. Insérer un ou plusieurs symboles et accepter ou changer t_i
   L'algorithme analyse présentement t_i qui équivaut à **c** et faisant partie de la chaîne initiale non corrigée **a c**. L'item analysé est $[T \rightarrow \mathbf{a} \bullet \mathbf{b}\ \mathbf{c}, f, c]$. Le premier préfixe est **b**. Le coût de ce préfixe est le coût de changer **c** par **b** (raisonnement identique au point 2 ci-haut). Le deuxième préfixe est **b c**. Le coût pour ce second préfixe est obtenu avec *incompleteCost**$(\mathbf{c}, \mathbf{b}\ \mathbf{c})$ et équivaut au coût d'insérer **b** (le coût d'insérer **b** plus le coût de changer **c** par **c** qui est 0). En considérant que le coût d'insérer **b** est inférieur au coût de changer **c** par **b**, l'opération d'édition pour ce cas est d'insérer **b**.

Lorsque le coût d'incomplétion *incompleteCost**$(t_i,$ *prefixes*$(\mathbf{t}\ \beta))$ est inférieur au coût de l'opération d'édition présentement stockée dans *findBestEditOperation*, cette dernière doit être changée afin de garder l'optimalité de la solution. Comme le coût d'incomplétion peut

générer différentes opérations d'édition en fonction du préfixe choisi, la procédure qui suit permet de déduire la prochaine opération d'édition devant être stockée. Cette procédure est utilisée par la fonction *incompleteCostOperation* (3.8).

La variable c utilisée dans la définition équivaut à $c = \mathit{incompleteCost*}(t_i, \mathit{prefixes}(\mathbf{t}\ \beta))$. La variable z représente le coût de mutation d'un terminal (précisement $z = \mathit{cost}(t_i, \mathbf{t})$). On considère également que $z \neq \mathit{cost}(\epsilon, \mathbf{t})$.

$$\mathit{incompleteCostOperation}(t_i, \mathbf{t}\ \beta) = \begin{cases} \mathit{EditOperation}(\mathit{Accept}, t_i, t_i, c) & \text{si } c = 0 \\ \mathit{EditOperation}(\mathit{None}, \epsilon, \epsilon, \infty) & \text{si } c = \infty \\ \mathit{EditOperation}(\mathit{Change}, t_i, \mathbf{t}, z) & \text{si } c = z \\ \mathit{EditOperation}(\mathit{Insert}, \epsilon, \mathbf{t}, c) & \text{autrement} \end{cases} \tag{3.8}$$

Un point important à mentionner est que les coûts simples ($\mathit{cost}(\ldots)$), composés ($\mathit{cost*}(\ldots)$), d'incomplétions ($\mathit{incompleteCost*}(\ldots)$) ainsi que le vecteur des opérations d'incomplétions ($\mathit{incompleteCostOperation}(t_i, \mathbf{t}\ \beta)$) peuvent tous être précalculés en amont de l'algorithme. En effet, ces coûts ne dépendent que de la grammaire et non de la phrase reçue en argument. Donc, une grande partie des calculs peuvent être faits avant le début de l'algorithme et réutilisés tant et aussi longtemps que la grammaire ne change pas.

Il est maintenant temps de présenter une exemple complet de l'algorithme qui sera utile à la compréhension. Nous allons reprendre la grammaire utilisée dans le chapitre 2 que nous avons recopié à la définition 3.9.

$$\begin{array}{lcl} P* & \rightarrow & \textbf{le}\ A\ S\ V\ N \\ A & \rightarrow & \textbf{beau} \mid \textbf{grand} \\ S & \rightarrow & \textbf{michel} \mid \textbf{dominique} \\ V & \rightarrow & \textbf{boit} \mid \textbf{voit} \\ N & \rightarrow & \textbf{du jus} \mid \textbf{de l'eau} \end{array} \tag{3.9}$$

Pour cette exemple, nous allons utiliser l'algorithme afin de corriger la phrase **le beu michel boit jus** qui contient quelques erreurs. Pour cet exemple, les coûts d'insertion, de mutation et de suppression sont respectivement 2, 3 et 3.4.

Au départ, l'algorithme ajoute l'item $[Z \rightarrow \bullet\ P\ \dashv, 0, 0]$ à l'ensemble s_0 dans sa phase d'initialisation. Juste après cette ajout, l'algorithme lance une phase de prédiction et de complétion. Cette phase de prédiction ajoutera l'item $[P \rightarrow \bullet\ \textbf{le}\ A\ S\ V\ N, 0, 0]$ à l'ensemble s_0 et calulera les coûts de complétion (les détails de cette opération peuvent être consultés à l'annexe A).

L'algorithme entre alors dans sa boucle principale qui se termine lorsque le terminal $\dashv$ aura été accepté. À ce point, l'algorithme est prêt à déterminer la meilleure opération d'édition pour le mot original t_i qui est **le**. Rappellons-nous que pour choisir l'opération d'édition, la fonction *findBestEditOperation* itère sur les items de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, c]$ dans l'ensemble s_j.

Présentement, l'algorithme est à l'itération $i = 0, j = 0$ et le seul item de la bonne forme dans l'ensemble s_0 est l'item $[P \rightarrow \bullet\ \mathbf{le}\ A\ S\ V\ N, 0, 0]$. Le coût d'incomplétion pour cette item (*incompleteCost**(**le**, *prefixes*(**le** A S V N))) est de 0 et l'opération choisie est d'accepter **le**. Puisqu'il ne pourra y avoir d'opération plus optimale, cette opération est retournée par la fonction *findBestEditOperation*.

Après avoir choisie l'opération, l'algorithme doit augmenter la liste u afin de bâtir la phrase corrigée. Le mot **le** sera ajouté à la liste u ($u \leftarrow u + [$*editOperation.terminal*$]$). On lance ensuite l'algorithme d'Earley originale sur la liste u. L'algorithme d'Earley fera en sorte d'ajouter l'item $[P \rightarrow \mathbf{le} \bullet A\ S\ V\ N, 0, 1]$ dans s_1 et prédira que les items $[A \rightarrow \bullet\ \mathbf{beau}, 1, 1]$ et $[A \rightarrow \bullet\ \mathbf{gand}, 1, 1]$ devront être ajoutés à l'ensemble s_1. Les variables i et j sont incrémentées de 1.

Il de temps de choisir la meilleure correction pour le mot original t_1 qui est **beu**. Dans l'ensemble s_i, deux items sont bloqués à un terminal soit les items $[A \rightarrow \bullet\ \mathbf{beau}, 1, 1]$ et $[A \rightarrow \bullet\ \mathbf{gand}, 1, 1]$. Dans les deux cas, le coût d'incomplétion est ∞ car le mot **beu** ne fait pas partie de l'ensemble T associé à la grammaire. C'est la même chose avec le coût de complétion qui sera également ∞ pour les deux items pour la même raison que pour les coûts d'incomplétion. Puisque l'opération de départ est de supprimer le mot **beu** et qu'aucune autre opération n'est plus optimale que celle-ci, elle sera retournée par la fonction *findBestEditOperation* comme opération optimale. Une note ici afin de mentionner que nous avons amélioré l'algorithme original afin qu'il soit en mesure de traiter des mots qui ne sont pas dans T. Cette modification est décrite à la section 3.1.1.

Puisque l'opération est de supprimer le mot **beu**, la variable i est incrémentée de 1 (devient $i = 2$) et l'algorithme doit choisir une nouvelle opération d'édition. Notez que la variable j n'a pas été incrémentée, on analyse donc toujours dans la fonction *findBestEditOperation* les items bloqués à un terminal qui se trouve dans l'ensemble s_j ($j = 1$). L'algorithme cherche alors la meilleure opération d'édition pour le mot t_2 qui est **michel**.

Commençons avec l'item $[A \rightarrow \bullet\ \mathbf{beau}, 1, 1]$. Le coût d'incomplétion est de 3 (changer **michel** vers **beau**). Puisque cette opération est plus optimale que l'opération initiale de supprimer **michel**, l'opération optimale devient *EditOperation*(*Change*, **michel**, **beau**, 3). La prochaine étape est de calculer le coût d'insérer **t** β, qui équivaut à **beau** car $\beta = \epsilon$, additionné au coût de complétion (*completionCosts*$[1][A, \mathbf{michel}]$).

Le coût d'insérer **beau** est de 2. Pour le coût de complétion, nous désirons expliquer un peu plus en détails ce à quoi il correspond.

Le coût de complétion peut être vu comme le coût de continuer l'analyse suivant le non-terminal A sachant qu'on traite le mot original **michel** et qu'il y a présentement 1 mot dans la phrase corrigée.

Le vecteur à deux dimensions *completionCosts* est utilisé pour obtenir un coût de complétion. La première dimension est indexée par le nombre de mots corrigés et la deuxième par une paire formée d'un non-terminal et d'un terminal. Par exemple, le coût de remplacer **t** par un préfixe suivant A après avoir corrigé j mots est donné par $completionCosts[j][A, \mathbf{t}]$. Le vecteur *completionCosts* est pré-calculé à chaque phase de prédiction par la fonction *predictCompletionCosts*.

Les détails pour cacluler le vecteur des coûts de complétion sont donnés à l'annexe A. Cependant, voyons tout de même comment le coût $completionCosts[1][A, \mathbf{michel}]$ est calculé.

On veut compléter ce qui suit le non-terminal A. Il y a une seule expansion menant à une analyse complète suivant A, l'expansion $S\ V\ N\ \dashv$. Pour qu'on soit en mesure de calculer le coût, il faut toujours que l'expansion commence par un terminal. Il y a deux expansions possibles : l'expansion **dominique** $V\ N\ \dashv$ et l'expansion **michel** $V\ N\ \dashv$.

Le coût de complétion pour l'expansion **dominique** $V\ N\ \dashv$ est de 3 car, pour continuer l'analyse, il faut changer **michel**, le mot original présentement analysé, par **dominique**. Pour l'expansion **michel** $V\ N\ \dashv$, le coût est de 0 car pour continuer, il faut accepter **michel** qui a un coût de 0.

Donc, le coût de complétion $completionCosts[1][A, \mathbf{michel}]$ est de 0 car c'est le coût le plus petit parmi tous les coûts calculés. Ceci veut donc dire que le coût $cost^*(\epsilon, \mathbf{beau}) + completionCosts[1][A, \mathbf{michel}]$ est de 2 (coût d'insérer **beau**). Comme ce coût est inférieur au coût de changer **michel** vers **beau** qui était de 3, l'opération optimale devient $EditOperation(Insert, \epsilon, \mathbf{beau}, 2)$.

Pour le second item bloqué, $[A \rightarrow \bullet\ \mathbf{gand}, 1, 1]$, le raisonnement est identique est l'opération la moins coûteuse sera $EditOperation(Insert, \epsilon, \mathbf{grand}, 2)$. Comme l'opération optimale courante a aussi un coût de 2, elle reste la meilleure est la fonction *findBestEditOperation* retourne $EditOperation(Insert, \epsilon, \mathbf{beau}, 2)$. On voit ici que l'algorithme aurait très bien pu retourner $EditOperation(Insert, \epsilon, \mathbf{grand}, 2)$ si cet item avait été analysé en premier. Quand on regarde la phrase, un humain comprend rapidement que **beu** devait être remplacé par **beau**. La modification que nous avons apporté à l'algorithme pour qu'il traite des mots inconnus à la grammaire devient très intéressante. Cette modification fera en sorte que l'algorithme déterminera qu'il doit changer **beu** en **beau** au lieu de supprimer **beu** puis d'ajouter un mot qui permet de continuer l'analyse.

Maintenant que l'opération est d'insérer **beau**, l'algorithme ajoute **beau** dans la liste de mots corrigés u et incrémente j de 1 qui devient 2. La variable i n'est pas incrémentée car on doit toujours analyser le mot **michel** dans la prochaine itération de la boucle. On lance alors encore une fois l'algorithme d'Earley original sur la phrase corrigée qui est maitenant **le beau**. Ceci fera avancer l'item $[A \rightarrow \bullet\ \mathbf{gand}, 1, 1]$ qui permettra ensuite de faire avancer l'item $[P \rightarrow \mathbf{le} \bullet A\ S\ V\ N, 0, 1]$. Au final, cela ajoutera les items $[S \rightarrow \bullet\ \mathbf{dominique}, 2, 2]$ et $[S \rightarrow \bullet\ \mathbf{michel}, 2, 2]$ à l'ensemble s_2.

Pour les deux prochaines itérations, l'algorithme analysera respectivement les mots t_2 et t_3 qui sont **michel** et **boit**. Ces deux mots seront acceptés de la même manière que l'algorithme à accepté le mot **le**. La phrase corrigée deviendra **le beau michel boit** et après que l'algorithme d'Earley ait fait son travail, on retrouvera dans l'ensemble s_4 les items $[N \rightarrow \bullet\ \mathbf{de\ l'eau}, 4, 4]$ et $[N \rightarrow \bullet\ \mathbf{du\ jus}, 4, 4]$.

L'algorithme doit maintenant choisir la meilleure opération pour le mot t_4 qui est **jus**. $[N \rightarrow \bullet\ \mathbf{de\ l'eau}, 4, 4]$, le coût d'incomplétion est de 3, soit changer **jus** en **de**. Il est en effet impossible d'insérer le mot **de** car la grammaire ne permet pas d'ajouter **de** avant **jus**. La raison est que la seule expansion suivant le non-terminal N qui permettrait un analyse complète est $\dashv$. Or, le mot présentement analysé est **jus**, il n'existe donc aucune opération permettant de passer de **jus** à $\dashv$. Le coût de complétion est donc ∞. L'opération optimale devient donc $\mathit{EditOperation}(\mathit{Change}, \mathbf{jus}, \mathbf{de}, 3)$.

Pour le second item analysé, $[N \rightarrow \bullet\ \mathbf{du\ jus}, 4, 4]$. Le coût d'incomplétion est de 2, soit insérer **du** car il possible d'insérer **du** avant **jus**. Le coût de complétion est ∞ pour la même raison que l'item précédent. L'opération optimale devient alors $\mathit{EditOperation}(\mathit{Insert}, \epsilon, \mathbf{du}, 2)$ et c'est cette dernière qui sera choisie comme meilleure opération d'édition pour l'analyse du mot **jus**.

L'algorithme ajoutera le mot **du** à la liste u, incrémentera la variable j de 1 et laissera telle quelle la variable i. L'algorithme d'Earley sera alors lancé. Ensuite, l'algorithme analysera encore le mot **jus** qui sera cette fois accepté. Finalement, l'algorithme d'Earley sera lancé une autre fois puis le mot suivant, le mot $\dashv$, sera accepté. La phrase corrigée sera alors **le beau michel boit du jus**.

3.1.1 Modifications à l'algorithme

Dans cette sous-section, deux modifications importantes à l'algorithme original d'Anderson-Backhouse sont décrites. La première modification est de permettre à l'algorithme de changer un terminal inconnu à la grammaire en un terminal connu par celle-ci. La deuxième consiste à modifier quelque peu le déroulement de l'algorithme de façon à obtenir la séquence d'opérations d'édition appliquées à la phrase corrigée.

La première modification a été effectuée à cause d'une limitation de l'algorithme. Dans l'implémentation originale, il n'est pas possible de changer un terminal inconnu de la grammaire par un terminal connu de celle-ci. Ce problème survient à cause de la nature des coûts précalculés. En effet, la seule opération de changement que l'algorithme peut produire survient dans la fonction *findBestEditOperation* lors du test du coût d'incomplétion *incompleteCost**$(t_i, \mathbf{t}\ \beta)$. Le coût d'incomplétion est précalculé à l'aide des terminaux et des non-terminaux de la grammaire seulement. Ceci veut donc dire qu'un terminal qui n'existe pas dans l'ensemble T de la grammaire ne sera pas pris en compte lors du précalcul des coûts d'incomplétion. Cette caractéristique est tout à fait logique car s'il fallait tenir compte des terminaux de la phrase entrée par l'utilisateur, les coûts d'incomplétion ne pourraient plus être précalculés.

Ceci pose un inconvénient majeur, particulièrement à cause du type d'erreurs que nous avons envisagées. Nous avons établi qu'une importante catégorie d'erreurs que nos algorithmes auraient à corriger serait des erreurs d'orthographe sur une ou deux lettres d'un mot. Par exemple, la grammaire reconnaît le mot **adresse**, mais l'entrée est plutôt **addresse**. Ce genre d'erreur serait corrigé à l'aide de deux opérations dans l'algorithme original. Une suppression du mot **addresse** suivi d'une insertion du mot **adresse**, si le mot **addresse** n'existe pas dans l'ensemble T. Pire, comme nous l'avons vu dans l'exemple, il se pourrait qu'il y ait plusieurs mots candidats pour l'insertion. Il y a donc une chance que ce ne soit même pas le bon mot qui soit inséré.

Pallier ce problème est quelque chose de relativement simple. La modification s'effectue au niveau de la fonction *findBestEditOperation*. Le truc est de calculer dynamiquement le coût d'incomplétion de *incompleteCost**$(t_i, \mathbf{t}\ \beta)$ lorsque $t_i \notin T$. Pour y arriver, on ajoute une condition dans la boucle principale de *findBestEditOperation* juste avant la première condition. Cette nouvelle condition vérifie si $t_i \notin T$. Si t_i ne fait pas partie de T, alors on calcule le coût d'incomplétion de *incompleteCost**$(t_i, \mathbf{t}\ \beta)$ et on stocke le résultat dans le vecteur qui contient les coûts d'incomplétion précalculés. Ceci permet alors de changer un mot inconnu en un autre qui, lui, fait partie de l'ensemble T.

Il faut également modifier la fonction *cost* afin qu'elle soit en mesure de changer un mot qui ne serait pas dans T en un autre mot de l'ensemble T. Cette modification est également très simple à mettre en place.

De plus, pour éviter de changer des mots qui sont éloignés l'un de l'autre en terme de correction, nous avons modifié quelque peu les valeurs de base des coûts. En effet, avec des coûts fixes, il arrivait qu'un mot tel que **deux** était changé en **cinq** bien que les deux mots n'ont aucune lettre en commun. L'opération que nous souhaitions dans ces cas était de supprimer **deux** puis d'insérer un autre terminal plus approprié. Pour y parvenir, nous avons

dynamisé quelque peu le coût de mutation et ajusté le coût de suppression.

Au départ, le coût de suppression était de 4 et le coût de mutation de 3. Ceci avantageait à tout coup la mutation dans les cas où cette opération était possible. Ce que nous avons fait a été de dynamiser le coût de mutation en ajoutant au coût de base la distance normalisée de Levenshtein (Levenshtein, 1966) entre les deux mots. La distance normalisée se situe toujours entre 0.0 et 1.0. La définition 4.2 définit la manière de normaliser la distance de Levenshtein.

Dans sa version originale, une distance de 0.0 équivaut à une correspondance parfaite entre les deux chaînes et une distance de 1.0 représente une différence maximale. Pour notre projet, nous avons inversé la distance normalisée de telle sorte qu'une valeur de 1.0 représente deux chaînes identiques et 0.0 le contraire. Cette mesure est appelée la similitude dans le paragraphe qui suit.

Avec le terminal **deux** et le terminal **cinq**, la similitude calculée est 0.0 et le coût de mutation est 3.0. Avec le terminal **six** et **dix**, la similitude obtenue est 0.66667 et le coût de mutation devient 3.66667. Ceci n'est toutefois pas encore suffisant pour permettre de favoriser une opération de suppression lorsque deux chaînes sont diamétralement opposées. En effet, le coût 3.0, qui dénote une incompatibilité maximale, est toujours inférieur à 4.0, qui est le coût de suppression. Nous avons donc également ajusté le coût de suppression de telle sorte que l'opération de mutation soit choisie pour deux chaînes près l'une de l'autre et que l'opération de suppression soit favorisée dans les cas où les chaînes sont relativement éloignées.

Nous avons établi le coût optimal de suppression à 3.4. Ce coût indique que pour qu'une opération de mutation soit favorisée, les deux chaînes doivent être semblables à plus de 40%. Ce coût a été trouvé grâce à des résultats expérimentaux. Nous avons lancé ensuite l'algorithme d'Anderson-Backhouse avec différentes valeurs de coût de suppression en utilisant le corpus d'évaluation du chapitre 5. Pour chaque valeur de coût de suppression différente, nous avons noté la moyenne des valeurs de Levensthein et de Jaccard entre la phrase corrigée par l'algorithme et la phrase du corpus. La définition 4.3 offre une définition formelle de la distance de Jaccard.

Le tableau 3.2 donne les résultats obtenus en fonction des différentes valeurs de coût de suppression.

Tableau 3.2 Résultats pour trouver la meilleure valeur de coût de suppression afin d'optimiser le taux de correction

	3.25	3.34	3.4	3.5	3.75	4.0
Jaccard	0.7475	0.7462	0.7492	0.7320	0.5323	0.6368
Levensthein	0.8854	0.8848	0.8865	0.8724	0.7502	0.8121

En regardant le tableau 3.2, on voit rapidement que les deux derniers coûts (3.75 et 4.0) sont largement surclassés par les autres possibilités. Des quatre valeurs restantes, le coût 3.4 est celui qui permet d'obtenir les meilleurs résultats. C'est donc ce coût que nous avons choisi comme coût de suppression dans notre algorithme modifié d'Anderson-Backhouse.

La deuxième modification provient d'un besoin que nous avions pour le projet de recherche. En effet, nous devons avoir accès à la façon dont la phrase corrigée a été bâtie et non pas seulement la phrase corrigée. Pour parvenir à obtenir la façon dont la phrase a été corrigée, nous avons besoin d'avoir la séquence d'opérations d'édition appliquées par l'algorithme à la phrase originale.

Ceci est relativement aisé à mettre en place puisque l'algorithme possède un endroit centralisé où les opérations d'édition sont choisies : la fonction *findBestEditOperation*. Également, il n'y a qu'un seul endroit où l'opération est appliquée, soit dans la fonction *recognize*.

L'idée est donc de modifier légèrement la fonction *recognize* pour que cette dernière garde en mémoire chaque opération retournée par la fonction *findBestEditOperation*. À la fin de la fonction *recognize* modifiée, en parcourant la liste, on obtient la séquence des opérations d'édition utilisée.

Les modifications ont été effectuées à trois endroits dans la fonction *recognize* (3.5). Au début de la fonction, la variable *operations* est initialisée à la liste vide ([ ]). Ensuite, les deux autres modifications surviennent immédiatement après un appel à la fonction *findBestEditOperation*. Dans les deux cas, l'idée est d'ajouter la nouvelle opération d'édition choisie à la fin de la liste *operations*. L'opérateur + concatène deux listes. De cette façon, toutes les opérations d'édition sont ajoutées à la liste à chaque fois qu'elles sont sélectionnées.

Algorithme 3.5 Modification à la fonction *recognize* de l'algorithme d'Anderson-Backhouse

```
initialize()
operations ← [ ]
tant que u_j ≠ ⊣ faire
    editOperation ← findBestEditOperation()
    operations ← operations + [editOperation]
    si editOperation.type est None alors
        exit(Aucune correction possible)

    tant que editOperation.type est Delete faire
        i ← i + 1
        editOperation ← findBestEditOperation()
        operations ← operations + [editOperation]

    Le reste de la fonction est identique...
```

3.2 Algorithme de Lyon

L'algorithme de Lyon a été présenté pour la première fois en 1974 dans le journal « *Communications of the ACM* » sous le nom (Lyon, 1974) « *Syntax-directed least-errors analysis for context-free languages : a practical approach* ».

Cet algorithme d'analyse syntaxique, dont les fondations reposent sur l'algorithme d'Earley (Earley, 1970), permet de reconnaitre des phrases contenant une ou plusieurs erreurs, et ce, en proposant toujours la correction contenant le moins d'erreurs possible. Utilisant les concepts de la programmation dynamique tels que proposés par Bellman, cet algorithme est qualifié de correcteur global. La correction est dite globale, car des décisions antérieures peuvent être réévaluées advenant qu'une meilleure correction soit trouvée plus loin dans le déroulement de l'algorithme.

D'une manière brève, l'idée de l'algorithme consiste à explorer toutes les possibilités d'erreurs. Un chemin ayant moins d'erreurs écrase les chemins comportant plus d'erreurs. Les chemins sont modélisés en utilisant des items d'Earley qui offrent une façon élégante de les encoder facilement. Les items sont augmentés d'un compteur qui stocke le nombre d'erreurs depuis le début du chemin complet. À la fin de l'algorithme, on récupère l'item d'Earley final et à l'aide de son compteur d'erreurs, on connaît le nombre d'erreurs que comportait la phrase entrée par l'utilisateur. À l'aide de petites modifications aux structures de données, il est également possible de reconstruire l'ensemble des corrections effectuées sur une phrase en utilisant les ensembles d'Earley remplis par l'algorithme.

L'algorithme effectue toujours sa correction en utilisant la phrase originale sans jamais la modifier. Plutôt que d'essayer de corriger la phrase directement, l'algorithme utilise des hypothèses d'erreurs au niveau des items d'Earley. Il relâche les contraintes strictes de la phase de scan de l'algorithme original en appliquant ces hypothèses d'erreurs aux items.

Il est en mesure de traiter trois hypothèses d'erreur distinctes. La première est l'hypothèse de suppression d'un terminal, c'est-à-dire que le mot n'est pas présent dans la phrase originale. On assume que le mot doit être inséré pour corriger la phrase. Suit ensuite la mutation d'un terminal, où un mot dans la phrase a été changé par un mot qui n'est pas dans la grammaire. On assume qu'il faut le changer dans la phrase originale par le mot de la grammaire choisi pour corriger la phrase. Et finalement l'insertion d'un terminal, qui survient lorsqu'un mot est de trop dans la phrase. On assume qu'il faut supprimer le mot de la phrase originale pour corriger la phrase. Dans chacun des cas, un coût est associé. Dans la cadre de notre implémentation, un coût fixe de 1 est associé à toutes les hypothèses d'erreur. Également, l'acceptation d'un terminal est modélisée dans l'implémentation à l'aide d'une hypothèse d'acceptation qui a toujours un coût de 0.

Comme pour l'algorithme de base d'Earley, l'algorithme d'analyse syntaxique robuste de Lyon reçoit en arguments deux éléments : une grammaire CFG $G = (N, T, P, S)$ ainsi qu'une liste de mots $t_0, t_1, \ldots, t_{n-2}, t_{n-1}$ où n est le nombre de mots dans la phrase. Le dernier mot de la liste t_{n-1} doit toujours être le symbole $\dashv$, qui représente la fin de phrase. Ce mot est important dans l'algorithme car il définit où doit s'arrêter l'analyse. Arrivé à ce point, on assume que l'analyse est terminée et qu'il n'y a plus d'hypothèse d'erreur à considérer. En effet, il n'est possible ni de supprimer ni de muter le symbole $\dashv$ en un autre terminal.

La première modification à apporter aux structures de données se situe au niveau des items d'Earley. Dans l'algorithme de Lyon, les items d'Earley sont utilisés pour définir des chemins dans un arbre syntaxique. Pour remplir le critère d'optimalité, il est impératif de pouvoir différencier deux items qui sont au même endroit en terme de chemin, mais qui n'ont pas le même nombre d'erreurs. Afin de réaliser la différenciation, l'item d'Earley est augmenté d'un compteur d'erreurs e. Ce compteur permettra de garder en mémoire le nombre d'erreurs hypothétiques attachées à l'item courant.

$$\begin{aligned} &\text{Item original :} && [A \to \alpha \bullet \beta, f, c] \\ &\text{Item augmenté :} && [A \to \alpha \bullet \beta, f, c, e] \end{aligned} \tag{3.10}$$

L'item $[A \to \alpha \bullet \beta, 0, 0, 0]$ n'est pas équivalent à l'item $[A \to \alpha \bullet \beta, 0, 0, 2]$ puisque dans le second cas, le compteur de l'item est 2 et non 0. À noter que dans le cas d'un reconnaisseur de phrases, savoir quelles sont les hypothèses en question n'a pas d'impact puisque le reconnaisseur veut seulement savoir si la phrase peut être reconnue ou non. Comme le projet se

concentre sur l'analyse syntaxique plutôt que sur la reconnaissance seule, nous avons pallié ce problème en gardant également dans l'item d'Earley la liste des hypothèses d'erreurs qui ont été explorées. Nous aborderons plus en profondeur cet aspect dans la section 3.2.1.

La deuxième modification importante au niveau des structures de données est appliquée à l'ensemble d'Earley. La modification en question consiste à ajouter une nouvelle fonction permettant d'ajouter un élément à l'ensemble. Cependant, cette nouvelle fonction se distingue de la fonction usuelle d'addition dans un ensemble dans la manière de déterminer si un élément doit être ajouté ou non. L'algorithme de Lyon retourne toujours la correction contenant le moins d'erreurs possible. Pour y arriver, l'algorithme doit s'assurer que chaque item d'Earley se trouvant dans l'ensemble est l'item contenant le moins d'erreurs parmi tous les items d'Earley similaire. Un item d'Earley est dit similaire à un autre si toutes les propriétés des deux items sont équivalentes à l'exception du compteur d'erreurs, qui peut être équivalent ou non. Donc, si deux items ont la même production, le même *dotOffset*, le même *currentIndex* et le même *earlierIndex*, ils sont considérés comme similaires, peu importe si leurs compteurs d'erreurs e sont identiques ou non.

Afin d'alléger les définitions subséquentes faisant appel à la notion de similarité entre deux items, nous utiliserons le symbole d'approximation ($\approx$) pour dénoter que deux items sont similaires. Par exemple, en reprenant l'exemple présenté un peu plus haut, les deux items suivants sont considérés comme étant similaires, puisque toutes les propriétés de l'item à l'exception du compteur d'erreurs sont équivalentes :

$$[A \rightarrow \alpha \bullet \beta, 1, 2, 5] \approx [A \rightarrow \alpha \bullet \beta, 1, 2, 8]$$

À l'inverse, les deux paires qui suivent ne sont pas considérées comme similaires. La première, parce que le *dotOffset* de l'item de droite n'est pas au même endroit que celui de l'item de gauche et la deuxième, parce que la production n'est pas la même pour les deux items :

$$[A \rightarrow \alpha \bullet \beta, 0, 1, 1] \not\approx [A \rightarrow \alpha\ \beta \bullet, 0, 1, 2]$$
$$[B \rightarrow \alpha \bullet \beta, 0, 1, 1] \not\approx [A \rightarrow \alpha \bullet \beta, 0, 1, 2]$$

Avec cette définition de similarité, il est maintenant possible de définir plus formellement la nouvelle fonction d'addition à un ensemble d'Earley que nous nommerons *addRestricted* (3.11). Cette fonction prend en argument un ensemble d'Earley ainsi qu'un item d'Earley et retourne un ensemble d'Earley possiblement modifié. Deux cas de figure doivent être envisagés pour cette fonction. Le premier se produit lorsqu'aucun item similaire à l'item à ajouter (i_a) ne se trouve déjà dans l'ensemble. Alors l'item est automatiquement ajouté à l'ensemble. Le deuxième survient dans le cas contraire, c'est-à-dire qu'un item similaire (i_s) est déjà présent

dans l'ensemble. Dans ce cas, on compare le compteur d'erreurs de l'item similaire avec le compteur d'erreurs de l'item devant être potentiellement ajouté. Si $i_a.e < i_s.e$, alors l'item déjà présent est enlevé de l'ensemble et remplacé par le nouvel item. Dans le cas où $i_a.e \geq i_s.e$, aucune action n'est prise et l'item n'est pas ajouté à l'ensemble.

$$addRestricted(S, i_a) = \begin{cases} S \cup \{i_a\} & \text{si } \nexists\, i_s \in S \mid i_s \approx i_a \\ (S \setminus \{i_s\}) \cup \{i_a\} & \text{si } \exists\, i_s \in S \mid i_s \approx i_a \wedge i_a.e < i_s.e \\ S & \text{autrement} \end{cases} \tag{3.11}$$

Rappelons-nous que l'algorithme original d'Earley contient cinq fonctions principales : *recognize*, *computeSet*, *predict*, *scan* et *complete*. Chacune de ces cinq fonctions devra être modifiée afin de devenir robuste aux erreurs de syntaxe. Également, la séquence des appels aux différentes fonctions est légèrement différente de celui de l'algorithme d'Earley.

La fonction *recognize* (3.6) est le coeur de l'algorithme et est responsable du contrôle des autres fonctions. Sa première tâche est d'initialiser les ensembles d'Earley et l'item de départ. Tout comme dans l'algorithme original, nous avons besoin d'initialiser $n+1$ ensembles d'Earley vides. Les ensembles sont numérotés $s_0, s_1 \ldots s_n$ où n est le nombre de mots. Pour faire l'initialisation, la production $Z \rightarrow S \dashv$, où S est la production de départ, est ajoutée à la grammaire. L'item $[Z \rightarrow \bullet\ S \dashv, 0, 0, 0]$ est ensuite ajouté à l'ensemble s_0.

Après l'initialisation, la fonction démarre le peuplement des ensembles d'Earley. Pour ce faire, la fonction effectue une boucle de 0 à $n-1$ et à chaque itération, définit la variable i représentant le numéro d'itération puis appelle la fonction *computeSet*. Grâce à la variable i, il est possible d'obtenir le mot en cours d'analyse t_i, l'ensemble à peupler s_i ainsi que l'ensemble suivant celui courant s_{i+1}.

Algorithme 3.6 Fonction *recognize* de l'algorithme de Lyon

pour $i = 0$ **à** n **faire**
    $s_i \leftarrow$ EarleySet()

$P \leftarrow P \cup \{Z \rightarrow S \dashv\}$
$s_0 \leftarrow addRestricted(s_0, [Z \rightarrow \bullet\ S \dashv, 0, 0, 0])$

pour $i = 0$ **à** $n - 1$ **faire**
    computeSet()

La fonction suivante, *computeSet* (3.7), est responsable de peupler les différents ensembles

d'Earley. Elle fait appel aux fonctions *predict*, *scan* et *complete* afin de réaliser son travail. Ces trois fonctions contiennent d'ailleurs la majeure partie des modifications à l'algorithme original et qui permettent à l'algorithme de Lyon d'être robuste aux erreurs.

La fonction commence par vérifier si l'itération actuelle n'est pas l'itération 0. Dans le cas où $i = 0$, on n'applique pas de phase initiale de scan ni de phase de complétion.

Donc quand $i \neq 0$, la fonction applique une première phase de scan sur tous les items de l'ensemble s_i dont le *earlierIndex* n'est pas équivalent à i. On scan donc tous les items qui ne sont pas originaires de l'ensemble courant. On applique ensuite une phase de complétion en appelant la fonction *complete*.

Puis, peu importe la value de i, la fonction appelle les fonctions *predict* et *scan*. Cette nouvelle phase de scan s'effectue cette fois sur tous les items dont le *earlierIndex* est équivalent à i. Ces deux dernières fonctions sont en quelque sorte responsables de préparer le terrain pour la prochaine itération de la fonction *computeSet*. Elles ajoutent aux ensembles s_i et s_{i+1} les items qui seront utilisés lors des prochaines phases de l'algorithme.

Algorithme 3.7 Fonction *computeSet* de l'algorithme de Lyon

si $i \neq 0$ **alors**
    scan($\{item \mid item$ **de la forme** $[A \rightarrow \alpha \bullet \beta, f, c, e]$ **dans** $s_i \wedge f = i \wedge \beta \neq \epsilon\}$)
    complete()

predict()
scan($\{item \mid item$ **de la forme** $[A \rightarrow \alpha \bullet \beta, f, c, e]$ **dans** $s_i \wedge f \neq i \wedge \beta \neq \epsilon\}$)

La fonction *predict* (3.8), contrairement à la fonction originale de l'algorithme d'Earley, doit prédire tous les cas de figure possibles. Donc, pour offrir la meilleure correction, on doit faire en sorte que toutes les productions de la grammaire soient ajoutées lors de la phase de prédiction.

Avec ce but en tête, la fonction *predict* devient très simple au niveau de l'implémentation. Il suffit d'aller chercher toutes les productions de la grammaire et de créer un item d'Earley initial pour chacune d'entre elles.

Algorithme 3.8 Fonction *predict* de l'algorithme de Lyon

pour tout *production* **de la forme** $A \rightarrow \alpha$ **dans** $P \wedge \alpha \neq \epsilon$ **faire**
    $s_i \leftarrow$ *addRestricted*($s_i, [A \rightarrow \bullet\ \alpha, i, i, 0]$)

La fonction *scan* (3.9) reçoit un ensemble d'items devant être scannés. Chaque item est de la forme $[A \rightarrow \alpha \bullet \beta, f, c, e]$ où α peut être vide et β ne peut l'être. Tout comme

dans l'algorithme original, la fonction est responsable de faire avancer le *dotOffset* des items d'Earley en fonction du mot t_i présentement analysé. Dans l'algorithme de Lyon, elle devient également responsable des différentes hypothèses d'erreur qui doivent être évaluées pour fournir une tolérance aux erreurs. Pour y parvenir, la fonction joue avec le *dotOffset* et le compteur d'erreur des items dans le but de prendre en considération toutes les hypothèses pouvant être traitées.

Il est bon de rappeler que pour l'algorithme de Lyon, l'ajout d'un item à un ensemble d'Earley est toujours effectué en utilisant la fonction *addRestricted* qui traite l'ajout d'une manière particulière. Donc, lorsque nous parlons d'ajout dans un ensemble lors des descriptions qui suivent, nous faisons toujours référence à cette fonction en particulier.

Considérons premièrement le cas où β est de la forme $\beta = \mathbf{t}\ \gamma$ où γ peut être vide. Dans cette situation, la fonction *scan* traite quatre hypothèses. Notez que les hypothèses ne sont pas mutuellement exclusives.

Une hypothèse est prise en compte lorsque β remplit certaines conditions particulières. Si les conditions sont rencontrées, la fonction crée un nouvel item, basé sur celui présentement analysé. Ce nouvel item est légèrement modifié et ajouté soit à l'ensemble s_i soit à l'ensemble s_{i+1}. Le choix de l'ensemble et de la modification à apporter au nouvel item dépends de l'hypothèse traitée. Voir l'algorithme 3.9 pour une description formelle des conditions pour chaque hypothèses et des actions prisent dans chaque cas.

Regardons maintenant le cas où β est plutôt de la forme $\beta = B\ \gamma$ où B est un non-terminal et γ est possiblement vide. L'item analysé sera alors de la forme $[A \rightarrow \alpha \bullet B\ \gamma, f, c, e]$ et la fonction *scan* doit traiter l'hypothèse que B ait été supprimé.

Pour y parvenir, la fonction ajoute l'item $[A \rightarrow \alpha\ B \bullet \gamma, f, i, e + d]$ dans l'ensemble s_i où d est le coût minimal pour supprimer le non-terminal B. Le coût d est équivalent à la quantité de terminaux dans la dérivation la plus petite de B. Ces coûts peuvent être précalculés dans l'initialisation de l'algorithme. Ils sont équivalents à la définition $cost^*(\epsilon, \{B\})$ (3.2), soit la définition de coût d'insertion présentée dans l'algorithme d'Anderson-Backhouse.

Finalement, dans le cas où la fonction traite une hypothèse de suppression, soit d'un terminal ou d'un non-terminal, l'item ajouté à l'ensemble s_i doit être rescanné. En effet, plusieurs terminaux en séquences pourraient avoir été supprimés, il faut donc rescanner ces items à chaque fois qu'une telle hypothèse est envisagée.

La fonction *complete* (3.10) de l'algorithme de Lyon est la dernière que nous abordons dans ce chapitre. Tout comme son pendant de l'algorithme original, cette fonction a la responsabilité de faire avancer le *dotOffset* d'un item bloqué à un non-terminal. Pour débloquer les items, elle cherche les items complétés dans l'ensemble courant s_i.

Afin de parvenir à couvrir tous les items bloqués, la fonction effectue une boucle de $j = i-1$

Algorithme 3.9 Fonction *scan* de l'algorithme de Lyon

Entrée(s) : *items*, un ensemble d'item d'Earley
  pour tout *item* **dans** *items* **faire**
    si *item* **de la forme** $[A \to \alpha \bullet \mathbf{t}\ \beta, f, c, e]$ **alors**
      // Acceptation du mot t_i
      si $\mathbf{t} = t_i$ **alors**
        $s_{i+1} \leftarrow addRestricted(s_{i+1}, [A \to \alpha\ \mathbf{t} \bullet \beta, f, i+1, e])$

      // Mutation du mot t_i
      si $\mathbf{t} \neq t_i$ **et** $\mathbf{t} \neq \dashv$ **et** $t_i \neq \dashv$ **alors**
        $s_{i+1} \leftarrow addRestricted(s_{i+1}, [A \to \alpha\ \mathbf{t} \bullet \beta, f, i+1, e+1])$

      // Suppression du mot t_i
      si $\mathbf{t} \neq \dashv$ **alors**
        $s_i \leftarrow addRestricted(s_i, [A \to \alpha\ \mathbf{t} \bullet \beta, f, i, e+1])$
        $scan(\{[A \to \alpha\ \mathbf{t} \bullet \beta, f, i, e+1]\})$

      // Insertion d'un nouveau mot avant t_i
      si $t_i \neq \dashv$ **alors**
        $s_{i+1} \leftarrow addRestricted(s_{i+1}, [A \to \alpha \bullet \mathbf{t}\ \beta, f, i+1, e+1])$

    sinon si *item* **de la forme** $[A \to \alpha \bullet B\ \beta, f, c, e]$ **alors**
      $d \leftarrow$ coût minimal pour supprimer B
      $s_i \leftarrow addRestricted(s_i, [A \to \alpha\ B \bullet \beta, f, i, e+d])$
      $scan(\{[A \to \alpha\ B \bullet \beta, f, i, e+d]\})$

jusqu'à 0. Pour chaque itération, la fonction recherche dans l'ensemble s_j tous les items de la forme $[A \rightarrow \alpha \bullet B\,\beta, f, j, e_b]$ où B est un non-terminal et α et β peuvent être vides. Ensuite, pour chaque item bloqué, elle cherche dans l'ensemble s_i les items complétés qui seraient en mesure de débloquer l'item. La fonction recherche des items de la forme $[B \rightarrow \gamma\ \bullet, j, i, e_c]$ dans l'ensemble s_i. S'il y a plusieurs items complétés, l'algorithme choisit celui ayant la plus petite valeur e_c. Puisque seule la valeur e_c est requise pour permettre d'avancer l'item bloqué, la présence de plus d'un item complété avec la même valeur e_c n'est pas un problème. Dans le cas où aucun item complété n'est trouvé, la fonction passe au prochain item bloqué.

À ce point, la fonction a un item bloqué ainsi qu'une valeur minimale e_c qui correspond à la valeur du compteur d'erreurs de l'item complété. L'item $[A \rightarrow \alpha\ B\ \bullet\ \beta, f, i, e_b + e_c]$ est alors ajouté à l'ensemble s_i, toujours à l'aide de la fonction *addRestricted*. Finalement, si la fonction *addRestricted* ne retourne pas l'ensemble original, une phase de scan est appliquée sur l'item $[A \rightarrow \alpha\ B\ \bullet\ \beta, f, i, e_b + e_c]$. Cette phase de scan est requise, car la fonction *scan* ne sera pas rappelée une nouvelle fois pour l'ensemble s_i, ce qui implique que des hypothèses d'erreurs pourraient être manquées si on ne refait pas un scan sur l'item ajouté.

Rendue à ce stade, la fonction a terminée de trouver des items complétés. Elle doit alors recommencer le même processus puisque de nouveaux items ont possiblement été ajoutés soit par le compléteur, soit par la phase de scan. Ces items, bloqués ou complétés, pourraient peut-être produire une nouvelle solution plus optimale. Cette boucle se termine lorsque l'ensemble s_i n'est plus mis à jour. On recommence alors avec les items bloqués de l'ensemble s_{j-1} et ainsi de suite jusqu'à l'ensemble s_0.

Algorithme 3.10 Fonction *complete* de l'algorithme de Lyon

pour $j = i - 1$ **à** 0 **faire**
  $itemBloques \leftarrow \{item \mid item$ **de la forme** $[A \rightarrow \alpha \bullet B\,\beta, f, j, e_b]$ **dans** $s_j\}$

  repéter
    $s \leftarrow s_i$
    pour tout $itemBloque$ **dans** $itemBloques$ **faire**
      $itemCompletes \leftarrow \{item \mid item \in s_i \wedge item$ de la forme $[B \rightarrow \gamma \bullet, j, i, e_c]\}$

      si $itemCompletes \neq \emptyset$ **alors**
        $e_c \leftarrow min(\{item.e_c \mid item \in itemCompletes\})$
        $s_i \leftarrow addRestricted(s_i, [A \rightarrow \alpha\ B \bullet \beta, f, i, e_b + e_c])$
        si s_i **a changé alors**
          $scan(\{[A \rightarrow \alpha\ B \bullet \beta, f, i, e_b + e_c]\})$
  jusqu'à $s = s_i$

3.2.1 Modifications à l'algorithme

L'algorithme de Lyon est à la base un algorithme qui détermine si une phrase peut être générée par une grammaire G même dans les cas où la phrase contient une ou plusieurs erreurs. Il permet également de savoir combien d'erreurs se trouvent dans la phrase, si erreurs il y a. Cependant, dans le cadre de notre projet de recherche, nous cherchons non seulement à savoir si la phrase peut être analysée, mais nous cherchons également à obtenir une version corrigée de la phrase afin de permettre une analyse subséquente.

Ceci est un point majeur qui n'est pas disponible au départ dans l'algorithme de Lyon. De par la nature des items d'Earley augmentés qui sont utilisés dans l'algorithme, il n'est pas possible de reconstituer les opérations d'édition appliquées à partir des items. En effet, le compteur d'erreurs présent dans l'item d'Earley augmenté est la seule information disponible en ce qui concerne les erreurs dans la phrase. Il est impossible à partir du seul compteur d'erreurs de déterminer les hypothèses d'erreurs appliquées à un item particulier.

Toutefois, l'information est disponible pendant l'exécution de l'algorithme et il suffit de la stocker afin de pouvoir la récupérer plus tard. L'algorithme de Lyon étant un dérivé de l'algorithme d'Earley, la reconstitution d'une phrase peut se faire une fois tous les ensembles d'Earley remplis. C'est-à-dire qu'on part de l'item final et on reconstruit l'arbre de syntaxe en suivant le chemin des différents items à travers les différents ensembles d'Earley. Le *earlierIndex* des items est alors utilisé pour déterminer dans quel ensemble a commencé un item précis. À partir de l'arbre de syntaxe, il est possible de retrouver la phrase originale entrée par l'utilisateur.

Pour parvenir à trouver la phrase corrigée par l'algorithme de Lyon, les opérations d'édition appliquées à chaque item sont requises. Nous avons donc repris le concept d'opérations utilisé par l'algorithme d'Anderson-Backhouse (3.1). Il y a donc quatre opérations d'édition possibles : *Accept*, *Delete*, *Insert* et *Change*. L'opération *None* n'est pas utilisée. Puisque dans l'algorithme de Lyon il est possible d'accepter ou supprimer un non-terminal, deux nouvelles opérations d'édition viennent se greffer aux quatre autres déjà connues. Ces opérations d'édition sont respectivement *AcceptRule* et *InsertRule*. Ces nouvelles hypothèses ne peuvent être appliquées que sur les non-terminaux d'un item.

La modification que nous avons apportée à l'algorithme de Lyon est d'ajouter un nouveau champ aux items d'Earley augmentés. En plus du compteur d'erreurs, nous avons ajouté une liste qui contient les opérations d'édition appliquées. Quand un nouvel item est créé, cette liste est vide. À chaque fois que l'algorithme fait avancer le *dotOffset* d'un item, une nouvelle opération d'édition est ajoutée à la liste. L'opération d'édition ajoutée dépend alors de la raison qui a fait avancer le *dotOffset* de l'item.

$$\begin{array}{ll} \text{Item augmenté :} & [A \rightarrow \alpha \bullet \beta, f, c, e] \\ \text{Item modifié :} & [A \rightarrow \alpha \bullet \beta, f, c, e, l] \end{array} \tag{3.12}$$

La liste des opérations d'édition n'est pas prise en compte pour comparer les items entre eux. C'est-à-dire que deux items similaires au niveau de la production, de la position dans la production et de l'index de départ, mais différents en ce qui a trait à la liste d'édition ou du compteur d'erreurs sont considérés comme étant similaires.

Cette particularité fait donc en sorte que l'algorithme ne peut retourner toutes les corrections pour un certain item. La raison est simple et provient du fonctionnement de l'algorithme. Lorsqu'un item est ajouté à un ensemble d'Earley qui contient déjà un item similaire, mais avec un compteur d'erreurs plus élevé, ce dernier est écrasé par le nouvel item qui a un compteur d'erreurs plus faible. De par la nature de l'opération d'équivalence, la liste d'édition n'est pas prise en compte. Ceci fait en sorte que l'item qui vient d'être écrasé avait possiblement une liste d'opérations d'édition différente. Cette liste est donc perdue et on ne sera plus en mesure de reconstituer cette série d'opérations d'édition plus tard dans le cas où l'item faisait partie d'une solution optimale.

Ceci n'empêche nullement l'algorithme de retourner la phrase contenant le moins d'erreurs. Seulement, toutes les possibilités d'opérations d'édition ne pourront être retrouvées pour un certain compteur d'erreurs avec la modification que nous avons mis en place.

Quatre fonctions doivent être modifiées pour tenir compte de la séquence d'opérations d'édition appliquée aux items d'Earley. Ce sont en fait toutes les fonctions qui ajoutent directement des items aux ensembles d'Earley. Les fonctions sont *recognize* (3.6), *scan* (3.9), *predict* (3.8) et *complete* (3.10).

Dans la fonction *recognize*, c'est l'ajout de l'item de départ dans s_0 qui a été changé. L'item ajouté est maintenant $[Z \rightarrow \bullet\ S \dashv, 0, 0, 0, [\,]]$ où $[\,]$ représente la liste vide. La fonction *predict* est modifiée de manière similaire, l'item prédit ajouté à l'ensemble s_i possède maintenant la liste vide lorsqu'il est créé.

Les fonctions *scan* et *complete* sont modifiées de manière quelque peu différente. Commençons par la fonction *complete*. Dans cette dernière, un item bloqué à un non-terminal est avancé lorsqu'un item complété pouvant le débloquer est disponible. L'item dont le *dotOffset* sera avancé verra sa liste d'opérations d'édition agrandie de telle sorte que l'opération de type *AcceptRule* y soit ajoutée. L'opérateur + sur les listes représente la concaténation de deux listes. Avec un item bloqué de départ de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, j, e_b, l_b]$, le nouvel item ajouté dans s_i sera $[A \rightarrow \alpha\ B \bullet \beta, f, i, e_b + e_c, l_b + [\mathit{EditOperation}(\mathit{AcceptRule}, B, B, e_c)]]$ où e_c est le compteur d'erreurs de l'item complété. Ce nouvel item sera également celui utilisé pour lancer la phase de scan associée à l'opération de complétion d'un item.

Finalement, la dernière fonction qui a besoin d'une modification pour supporter une liste d'opérations d'édition est la fonction *scan*. Cette dernière traite quatre cas possibles d'opérations d'édition pour l'item présentement analysé. Avec un item de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, c, e, l]$ dans la boucle de la fonction, voici les quatre cas possibles.

1. Accepter **t**
   L'item ajouté dans s_{i+1} sera $[A \rightarrow \alpha\ \mathbf{t} \bullet \beta, f, i+1, e, l + [\mathit{EditOperation}(\mathit{Accept}, \mathbf{t}, \mathbf{t}, 0)]]$.
2. Changer t_i en **t**
   L'item ajouté dans s_{i+1} sera $[A \rightarrow \alpha\ \mathbf{t} \bullet \beta, f, i{+}1, e{+}1, l{+}[\mathit{EditOperation}(\mathit{Change}, t_i, \mathbf{t}, 1)]]$.
3. Insérer t_i
   L'item ajouté dans s_i sera $[A \rightarrow \alpha\ \mathbf{t} \bullet \beta, f, i, e+1, l + [\mathit{EditOperation}(\mathit{Delete}, t_i, \epsilon, 1)]]$.
4. Supprimer **t**
   L'item ajouté dans s_{i+1} sera $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, i{+}1, e{+}1, l{+}[\mathit{EditOperation}(\mathit{Insert}, \epsilon, \mathbf{t}, 1)]]$.

Une petite note sur les deux derniers cas. Dans ces deux cas, l'opération d'édition est inversée. Prenons l'exemple où on traite l'hypothèse que le terminal **t** doit être supprimé. Cela veut donc dire qu'il est manquant dans la phrase originale. Il faudra donc l'ajouter à la phrase corrigée pour être en mesure de la reconstruire, l'opération est inversée par rapport à l'hypothèse. Cette logique s'applique également à l'hypothèse que t_i a été inséré, il faut donc le supprimer dans la phrase originale afin de reconstruite la phrase corrigée.

Le dernier cas à traiter survient lorsque l'item dans la boucle est de la forme $[A \rightarrow \alpha \bullet B\ \beta, f, c, e, l]$. Dans ce cas, l'algorithme doit faire l'hypothèse que le non-terminal B pourrait être absent. L'opération d'édition à appliquer est donc d'insérer le non-terminal B, car il sera manquant dans la phrase. Si on assume que d est le coût minimal pour supprimer le non-terminal B, alors l'item modifié ajouté à l'ensemble s_i est $[A \rightarrow \alpha\ B \bullet \beta, f, i, e+d, l + [\mathit{EditOperation}(\mathit{InsertRule}, \epsilon, B, d)]]$.

Cette série de modifications nous permet alors de reconstruire la phrase corrigée, car nous avons maintenant accès à la liste d'opérations d'éditions appliquée à chaque item des ensembles d'Earley.

3.3 Algorithme hybride

L'algorithme hybride que nous présentons dans cette section est un algorithme qui a été créé de toutes pièces dans le cadre du projet de maîtrise. Même s'il est de conception originale, l'idée principale derrière l'algorithme est quant à elle déjà connue depuis longtemps et a déjà été utilisée à plus d'une reprise. L'article (Mellish, 1989) ainsi que (Lee *et al.*, 1995) proposent notamment des systèmes reprenant cette idée. Le truc est de simplement utiliser

deux analyseurs en tandem. Le premier, standard et non robuste aux erreurs, s'exécute sur la phrase d'entrée jusqu'à ce qu'il rencontre une erreur. Au point d'erreur, le deuxième analyseur, qui lui est robuste aux erreurs, est lancé et réutilise les informations générées par le premier.

Dans notre implémentation, l'algorithme d'Earley est utilisé comme analyseur standard et l'algorithme de Lyon comme algorithme robuste aux erreurs. La raison qui nous a poussé à développer cet algorithme était d'améliorer le temps d'exécution de l'algorithme de Lyon. En effet, le plus gros défaut de l'algorithme de Lyon est de passer autant de temps sur les régions de la phrase ne contenant pas d'erreurs que sur les régions de la phrase qui en contiennent.

De par sa nature globale, l'algorithme de Lyon n'a d'autre choix que de prendre en considération toutes les possibilités de corrections possibles même lorsqu'une phrase est valide du début à la fin. Ceci est bien évidemment très coûteux puisque si la phrase ne contient qu'une seule erreur vers la fin de la phrase, tout le travail effectué par l'algorithme de Lyon au début de cette même phrase a de grandes chances d'être inutile.

Nous avons décidé d'explorer l'idée d'un algorithme hybride pour réduire le temps de calcul de l'algorithme de Lyon. L'algorithme est relativement simple puisque son développement s'articule autour de techniques permettant de faire le pont entre l'algorithme d'Earley classique et l'algorithme Lyon. D'une manière brève, l'algorithme d'Earley est exécuté en premier sur la phrase originale. Ce dernier peuple alors les ensembles d'Earley jusqu'à ce qu'il rencontre une erreur. Ensuite, l'algorithme de Lyon démarre, mais conserve les ensembles d'Earley déjà remplis puis termine ensuite l'analyse de la phrase.

L'algorithme hybride que nous proposons performe donc aussi bien que l'algorithme d'Earley dans le cas où la phrase ne contient aucune erreur. Par la suite, sa performance dépend de la position de la première erreur. Si le premier mot est invalide, alors l'algorithme hybride sera aussi lent que l'algorithme de Lyon classique. Si l'erreur se trouve à la fin de la phrase, il n'y aura pas de surcoût au temps de calcul pour ce qui se trouve avant l'erreur.

On peut voir l'algorithme hybride comme un algorithme de correction régionale où la région de correction s'étend du premier point d'erreur jusqu'à la fin de la phrase. Par exemple, dans le cas d'une phrase de n mots et où la première erreur de la phrase est détectée à e, alors la région de correction est $t_e, t_{e+1}, \ldots, t_{n-2}, t_{n-1}$. Cependant, l'algorithme que nous présentons n'a pas la prétention d'être un réel algorithme régional, auquel cas il faudrait limiter la longueur de la région afin d'être en mesure de réactiver l'algorithme d'Earley classique pour d'autres régions de la phrase.

Nous allons maintenant décrire plus en détails l'algorithme hybride que nous avons développé. Tout d'abord, une note sur le type d'item qui est utilisé dans cet l'algorithme. Puisque normalement l'algorithme d'Earley et l'algorithme de Lyon utilisent des items légèrement différents, il faut décider de la manière d'intégrer le tout. Le plus simple est d'utiliser les items

de l'algorithme de Lyon. Lorsque l'algorithme hybride sera en mode sans correction d'erreurs, l'algorithme ne tiendra pas compte du compteur d'erreurs. Ce dernier restera donc toujours à 0, ce qui est valide pour l'algorithme d'Earley. La seule différence notable est qu'il faudra tout même garder la trace des opérations d'édition qui ont été effecutées par l'algorithme pour chaque item.

L'algorithme hybride débute avec une version légèrement modifiée de la fonction *recognize* (3.11) de l'algorithme d'Earley (2.1). On modifie la boucle principale de cette fonction en y ajoutant une condition supplémentaire. Cette condition sert à vérifier que l'ensemble s_i a été peuplé avec au moins un item. Si l'ensemble s_i est vide, cela implique qu'une erreur est survenue et que le mot t_{i-1} n'a pas été reconnu. Quand ce cas survient, on assigne une variable avec la valeur de i. Cette variable, la variable k, sert alors de sentinelle et permet d'arrêter la boucle principale lorsque l'analyse classique a échouée. Lorsque la variable ne contient pas sa valeur initiale qui est -1, on appelle *recognizeFallback*, qui reçoit l'index où doit commencer l'analyse robuste. Cette fonction s'occupe alors de faire le pont entre l'algorithme d'Earley et l'algorithme de Lyon.

Algorithme 3.11 Fonction *recognize* de l'algorithme hybride

pour $i = 0$ **à** n **faire**
    $s_i \leftarrow$ EarleySet()

$P \leftarrow \cup \{Z \rightarrow S \dashv\}$
$s_0 \leftarrow s_0 \cup \{[Z \rightarrow \bullet\ S \dashv, 0, 0]\}$

$i \leftarrow 0$
$k \leftarrow -1$

repéter
    earleyComputeSet()
    si $i \neq 0$ **et** $s_i = \emptyset$ **alors**
        $k \leftarrow i$

    $i \leftarrow i + 1$
jusqu'à $i > n$ **ou** $k \neq -1$

si $k \neq -1$ **alors**
    recognizeFallback(k)

L'appel à la fonction *earleyComputeSet* dans la fonction *recognize* de l'algorithme 3.11 est un appel à la fonction *computeSet* de l'algorithme classique d'Earley (2.2). La fonction est

préfixée du mot *earley* pour différencier les fonctions de l'algorithme d'Earley des fonctions ayant le même nom dans l'algorithme de Lyon. Les fonctions de l'algorithme de Lyon sont quant à elles préfixées avec *lyon*. L'utilisation de la fonction *earleyComputeSet* permet de faire fonctionner l'algorithme d'Earley puisque toutes les autres fonctions de l'algorithme classique seront appelées via cette fonction.

Comme nous le mentionnions précédemment, la fonction *recognizeFallback* fait le pont entre l'algorithme d'Earley et celui de Lyon. Elle reçoit en argument k qui détermine l'indice du premier ensemble vide rencontré par la fonction *recognize* présentée ci-haut. La fonction *recognizeFallback* va ensuite démarrer une nouvelle boucle principale qui commence à l'index $k-1$. L'ensemble de départ sera celui qui précède le premier ensemble vide trouvé. Le fait de commencer avec l'ensemble s_{k-1} plutôt que le premier ensemble vide s_k permet de prédire des items avec le prédicteur de Lyon qui n'ont pas été prédit par le prédicteur de l'algorithme d'Earley. La fonction s'occupe ensuite d'appeler les différentes fonctions de l'algorithme de Lyon. D'une certaine manière, la fonction *recognizeFallback* (3.12) vient remplacer la fonction *recognize* de l'algorithme de Lyon (3.6).

Algorithme 3.12 Fonction *recognizeFallback* de l'algorithme hybride

Entrée(s) : k index du premier ensemble vide
  pour $i = k-1$ **à** $n-1$ **faire**
    lyonComputeSet()

L'appel à la fonction *lyonComputeSet* fait en sorte que l'algorithme de Lyon s'exécutera de l'ensemble s_{k-1} jusqu'à l'ensemble s_n.

La complexité de cet algorithme dans le pire cas est équivalente à l'algorithme de Lyon. Il survient quand le premier mot de la phrase est invalide. Dans un tel cas, l'algorithme de Lyon s'exécute de l'ensemble s_0 jusqu'à l'ensemble s_n, ce qui est identique à l'algorithme de Lyon. Dans le meilleure des cas, tous les mots de la phrase sont valides et la complexité est alors équivalente à l'algorithme d'Earley.

CHAPITRE 4

MÉTHODOLOGIE

Dans ce chapitre, on parle de la méthodologie qui a été élaborée et utilisée pour obtenir tous les résultats expérimentaux qui sont présentés dans la section résultats (5). On y parle des méthodes d'évaluation choisies, des mesures de performance, des grammaires utilisées pour l'évaluation ainsi que du corpus de phrases choisie pour chaque grammaire.

Pour parvenir à bien faire l'évaluation des algorithmes, il faut en premier lieu déterminer leurs usages subséquents. Des algorithmes utilisés pour analyser des centaines de milliers de pages web contenant plusieurs phrases ne seront pas évalués de la même façon que des algorithmes devant analyser une seule phrase de manière très rigoureuse. Deux usages principaux ont été envisagés pour les algorithmes syntaxiques robustes que nous avons évalués.

Le premier usage serait un outil graphique afin de comprendre pourquoi une phrase particulière ne fait pas partie du langage généré par une grammaire G. Cet outil permettrait au développeur de grammaires de déterminer les raisons pour lesquelles la phrase n'a pu être analysée correctement par un algorithme classique.

Le deuxième usage envisagé serait d'inférer des améliorations à une grammaire G afin d'augmenter sont taux de reconnaissance (nombre de phrases reconnues par rapport au nombre total de phrases dans un ensemble). On analyserait un ensemble comprenant entre 2 et 5000 phrases non reconnues par la grammaire G. Pour chacune d'entre elles, l'algorithme d'analyse syntaxique robuste trouverait la meilleure correction possible. Avec la meilleure correction pour chaque phrase, il suffirait ensuite de trouver les erreurs récurrentes et d'inférer les améliorations à apporter à la grammaire.

Le terme diagnostic des erreurs sera dorénavant utilisé pour faire référence à ces deux usages.

En prenant en considération les caractéristiques requises pour chaque usage envisagé, nous en sommes venu à la conclusion que la donnée la plus importante pour l'évaluation des algorithmes est la qualité des corrections effectuées. En effet, il est impératif que la correction soit d'une très grande qualité, car si elle est médiocre, le diagnostic des erreurs sera durement affecté. Ceci nous amène alors vers le problème de définir quantitativement la qualité des corrections d'erreurs. Concrètement, il faut au départ définir ce qu'est la meilleure correction possible, puis quantifier ce critère de manière à pouvoir comparer les différents algorithmes entre eux.

La meilleure correction possible d'une phrase est la correction respectant la syntaxe établie

par la grammaire et se rapprochant le plus possible de l'intention réelle de la personne ayant dit la phrase. Cette correction optimale est la cible visée par les algorithmes d'analyse syntaxique robuste. Si hypothétiquement nous avions accès à cette correction optimale, nous serions en mesure de comparer chaque algorithme avec cette correction étalon.

Pour parvenir à comparer une correction par rapport à une autre, nous avons tout d'abord établi un codage au niveau de la correction proposée par les algorithmes. Grâce à la liste des opérations d'édition qui a été ajoutée aux algorithmes syntaxiques robustes, chacun d'entre eux est en mesure de fournir une version corrigée de la phrase à partir des différents items qu'il a mis dans les ensembles d'Earley. Chacun des mots de cette phrase corrigée est encodée de telle sorte que chaque opération d'édition est représentée.

1. Un mot correct est laissé tel quel dans la version encodée (**correct**).
2. Un mot inséré par l'algorithme est encodé en suffixant le signe plus au mot inséré (**inséré+**).
3. Un mot supprimé par l'algorithme est encodé en suffixant le signe moins au mot supprimé (**supprimé-**).
4. Un mot muté par l'algorithme est encodé en plaçant le nouveau mot suivi d'une barre verticale (|) suivi de l'ancien mot mis entre parenthèses. (**nouveau|(ancien)**)

Le tableau 4.1 reprend la grammaire de la section 2 (qui est reproduite à la définition 4.1) et démontre un exemple du système d'encodage des opérations d'édition sur différentes phrases.

$$\begin{array}{lcl} P^* & \rightarrow & \textbf{le } A\ S\ V\ N \\ A & \rightarrow & \textbf{beau} \mid \textbf{grand} \\ S & \rightarrow & \textbf{michel} \mid \textbf{dominique} \\ V & \rightarrow & \textbf{boit} \mid \textbf{voit} \\ N & \rightarrow & \textbf{du jus} \mid \textbf{de l'eau} \end{array} \tag{4.1}$$

Tableau 4.1 Encodage des corrections proposées par les algorithmes

	Phrase originale	Version encodée
Mot correct	le beau michel boit du jus	le beau michel boit du jus
Mot inséré	le beau michel boit jus	le beau michel boit **du+** jus
Mot supprimé	le beau **brun** michel boit du jus	le beau **brun-** michel boit du jus
Mot muté	le beau michel boit du **jut**	le beau michel boit du **jus|(jut)**

Grâce à ce système d'encodage, il devient en effet aisé de comparer deux corrections entre

elles et déterminer la similitude entre les deux. Nous sommes alors en mesure d'établir une quantification entre la correction optimale d'une phrase et la correction proposée par un algorithme. Pour y parvenir, nous avons choisi deux mesures de similitudes bien établies, soit la distance de Levenshtein (Levenshtein, 1966) et la distance de Jaccard (Wikipedia, 2005).

La première est une mesure de similitude sur les chaînes de caractères permettant d'obtenir le nombre d'opérations requises pour transformer une chaîne en une autre. Nous avons adapté cette mesure pour qu'elle calcule le nombre d'opérations d'édition requises au niveau des mots pour transformer une phrase complète en une autre. La distance obtenue est ensuite normalisée et se situe alors entre 0.0 et 1.0. On normalise en divisant la valeur obtenue par le nombre de mots dans la phrase la plus longue. La distance normalisée est ensuite inversée de telle sorte qu'une valeur de 1.0 dénote une similarité parfaite entre deux phrases et qu'une valeur de 0.0 dénote une différence totale. Par exemple, la distance entre la phrase « le beau michel boit jut » et le mot « le beau michel boit du jus » est de 2 (un mot ajouté et un mot muté). Il peut y avoir au maximum 6 erreurs (nombre de mots dans la phrase la plus longue), on obtient donc une distance normalisée de 0.6666 (2/6) et finalement une distance normalisée inversée de 0.3334 ($0.3334 = 1.0 - 0.6666$).

$$L(A, B) = 1 - \frac{levenshtein(A, B)}{max(|A|, |B|)} \tag{4.2}$$

Nous avons également pris en compte une deuxième mesure dans notre plan d'expérimentation. Celle-ci, la distance de Jaccard, est plutôt une mesure de similitude entre deux ensembles. Nous avons choisi d'utiliser cette mesure principalement car elle ne prend pas en considération la position des mots dans la phrase, contrairement à la distance de Levenshtein.

Prenons deux ensembles, l'ensemble A formé des mots de la correction optimale et l'ensemble B celui formé des mots de la correction proposée. Cette mesure se calcule alors simplement via l'équation 4.3 qui dit que la distance est la proportion entre le nombre d'éléments ne se retrouvant pas dans les deux ensembles par le nombre d'éléments contenus dans l'union des deux ensembles.

$$J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{4.3}$$

Le fait que la distance de Jaccard ne fasse pas de distinction au niveau de la position des éléments dans un ensemble peut causer quelques soucis dans certains cas. Par exemple, les phrases « du jus » et « jus du » donnent une distance de Jaccard de 1.0 tout comme la phrase « du jus » et « du jus ». Malgré le fait que nous n'avons pas de chiffre précis, nos observations tendent vers le fait que ce comportement aberrant n'est pas un grave problème car il ne se produit pas très souvent.

Un autre problème vient des ensembles et des mots répétés. Dans une phrase, il peut y avoir plusieurs fois le même mot. Comme la distance travaille sur des ensembles, la répétition des mots n'est pas prise en compte. Pour pallier ce problème, nous utilisons des *multi-set* qui comptent le nombre d'éléments. De cette façon, une correction proposée contenant une seule fois un mot se trouvant 3 fois dans la correction optimale aura une distance de Jaccard plus faible qu'une autre contenant 2 fois le mot.

À ce point-ci, nous avons en main la méthodologie pour chiffrer quantitativement la différence entre deux corrections. Nous avons établi deux mesures de similitude dont les valeurs respectives se trouvent toutes deux entre 0.0 et 1.0.

Mais pourquoi avoir choisi une plage de valeur pour les mesures plutôt que simplement dire qu'une phrase corrigée correspond exactement à la correction optimale et ensuite calculer la proportion de corrections strictement identiques ?

La raison est plutôt simple et provient du fait que la correction optimale est souvent inatteignable. Ceci survient car il n'est pas toujours possible d'obtenir la correction représentant l'intention réelle de l'utilisateur. En effet, la meilleure correction possible parmi un ensemble de propositions peut être ambigüe.

Prenons l'exemple d'un utilisateur à qui on demande d'entrer son numéro de téléphone de 10 chiffres avec des mots. Il dit alors la phrase suivante : « cinq un quatre cinq cinq cinq un un un un deux ». Quel était réellement le numéro que voulait dire l'utilisateur ? Était-ce 514-555-1111 (enlever le dernier mot) ou bien 514-555-1112 (enlever le dernier « un ») ?

Dans notre cadre applicatif, les deux corrections seraient valables. Donc le problème avec une mesure précise de type vrai ou faux est qu'il y aurait des cas où la mesure de similitude serait de 0.0 alors qu'au fond, elle est très près de la correction optimale et la différence n'est dûe qu'à l'ambiguïté entre les deux corrections possibles.

De plus, sans autre information, il n'est pas possible d'affirmer quoi que ce soit sur la meilleure correction possible. Il arrive cependant que de l'information sémantique vienne aider à faire ce choix. La phrase suivante contient une telle information : « cinq un quatre cinq cinq cinq un un un un euh non deux ». Le fait d'avoir un « euh non » nous amène à nous dire que l'utilisateur s'est probablement trompé et a essayé de corriger son erreur.

Donc, en utilisant des mesures de similitude entre 0.0 et 1.0, nous avons la possibilité de dire qu'une correction proposée est près ou non de la correction optimale.

Puisque de l'information sémantique pourrait être utilisée pour déterminer la meillure correction possible, il a été convenu que la correction optimale sera fournie par un être humain et non une machine. La raison est qu'un humain peut extraire plus facilement l'information sémantique d'une phrase comparativement à ce qu'un ordinateur est en mesure de faire. Mais même un humain ne peut pas toujours déterminer avec certitude la correction optimale d'une

phrase. On ne parlera donc plus de correction optimale dans les paragraphes suivants, mais plutôt de correction de référence.

Bien sûr, un humain est naturellement subjectif. Pour éviter le plus possible un biais dans la correction de référence, il a été décidé que trois corrections de référence allaient être comparées à la correction proposée par chaque algorithme. Ces trois corrections seront produites par trois personnes différentes.

Comparer la performance des algorithmes en se basant sur une seule phrase provenant d'une seule grammaire serait un peu ridicule. Nous avons donc choisi de comparer les algorithmes grâce à plusieurs phrases provenant de plusieurs grammaires différentes. Nous avons choisi d'évaluer les différents algorithmes par rapport à 10 grammaires distinctes. Pour chacune de ces grammaires, 25 phrases ont été sélectionnées parmi un corpus de phrases recueilli spécialement pour la grammaire. Attardons-nous quelque peu sur la provenance de ces grammaires et de ces phrases.

L'entreprise d'accueil, *Nu Echo Inc.*, a produit au fil des années plusieurs applications utilisant la reconnaissance de la voix. Ces applications utilisent des grammaires pour définir la syntaxe attendue des réponses dites par les utilisateurs. On retrouve plusieurs types de grammaires différentes : des grammaires pour reconnaitre des adresses, des codes postaux, pour obtenir une confirmation, pour déterminer un département, etc. La compagnie possède également plusieurs outils de collecte de données et d'analyse de résultats très performants. Les outils de collecte permettent par exemple d'obtenir les phrases dites par des utilisateurs réels pour une grammaire particulière. L'outil est ensuite en mesure d'extraire le texte du segment audio. Un employé va ensuite parcourir l'ensemble des phrases retranscrites automatiquement et s'assurer que la transcription est bonne.

Il est maintenant temps d'élaborer plus en détail la manière dont ont été sélectionnées les 10 grammaires et les 25 phrases associées à chaque grammaire. Débutons par les phrases. Pour chaque ensemble de phrases associé à une grammaire, un groupement des phrases a été effectué. Ce groupement s'est effectué de la manière suivante. Un algorithme classique d'analyse syntaxique, dans notre cas l'algorithme d'Earley, utilise la grammaire associée à l'ensemble et analyse chaque phrase une par une. Si la phrase est reconnue par l'algorithme, alors la phrase est de type *In-Grammar* (ING) sinon, elle est de type *Out-Of-Grammar* (OOG). Ce premier regroupement nous donne donc deux catégories de phrases. C'est la catégorie de phrases OOG qui nous intéresse plus particulièrement car c'est dans cette catégorie que se retrouvent les phrases contenant des erreurs.

Cependant, cette catégorie est plutôt vaste et peut contenir des incongruités, comme dans le cas où un utilisateur donne son code postal alors qu'il devait donner sa date de naissance. Nous avons donc effectué un deuxième regroupement dans la catégorie de phrases OOG.

Nous avons utilisé un deuxième algorithme d'analyse syntaxique développé par la compagnie d'accueil. Cet algorithme, que nous appellons *chunker*, permet d'extraire les constituants reconnues d'une phrase OOG. Il est différent des algorithmes syntaxiques robustes, car il n'est pas en mesure de déterminer précisement où sont les erreurs ni la manière de les corrigées. Il ne peut donc pas faire de diagnostic d'erreurs.

Nous avons lancé le *chunker* sur chacune des phrases se retrouvant dans la catégorie OOG. Si le *chunker* est en mesure d'extraire des constituants valides de la phrase, la phrase est libellée comme *Incorrect*. Si le *chunker* ne retourne rien de valide, la phrase est libellée *Highly Incorrect*. Après cette deuxième phase de regroupement, nous obtenons trois catégories : les phrases ING que nous appellerons dorénavant les phrases *Correct*, les phrases *Incorrect* et les phrases *Highly Incorrect*.

Parmi ces trois catégories, c'est la deuxième qui nous intéresse plus particulièrement. En effet, c'est dans cette catégorie que se retrouvent les phrases qui ont le meilleur potentiel d'améliorer le taux de reconnaissance de la grammaire.

Il n'y a aucune phrase contenant des erreurs dans la première catégorie tandis que la dernière contient trop de phrases farfelues n'ayant bien souvent aucun lien avec la grammaire à laquelle elles sont rattachées. Les 25 phrases de l'ensemble de tests proviennent toutes de la catégorie *Incorrect*. Elles ont été choisies au hasard, à l'aide d'un programme informatique, en s'assurant qu'aucune phrase ne se retrouve deux fois dans l'ensemble.

Pour effectuer le choix des grammaires, nous avons tout d'abord filtré les grammaires dont la catégorie *Incorrect* ne contenait pas au moins 25 phrases. Celles-ci ne faisaient pas partie des grammaires pouvant être sélectionnées pour la comparaison. Nous avons ensuite essayé de choisir 10 grammaires différentes les unes par rapport aux autres ne traitant pas du même concept. Par exemple, une grammaire reconnaissant des dates de naissance traite du même concept qu'une grammaire qui reconnait une date d'activation.

Par la suite, nous avons également choisi des grammaires ayant des caractéristiques différentes. Particulièrement en ce qui concerne le nombre de terminaux, le nombre de non-terminaux, le nombre de productions et le nombre de productions anullables (c'est-à-dire $|\{p \in P \mid p \to \epsilon\}|$).

Comme les algorithmes ne sont pas dépendants de la langue utilisée, nous avons sélectionné sept grammaires en français et trois en anglais pour démontrer ce fait.

Tous ces choix ont été fait afin d'obtenir un ensemble représentatif de grammaires. Dans notre cadre applicatif, une infinité de grammaire différentes peuvent être produites. Nous avons donc décidé de choisir des grammaires utilisées dans plusieurs applications distinctes. C'était selon nous un bon moyen d'avoir des grammaires ayant une certaine importance dans notre cadre applicatif.

Le tableau 4.2 contient une description des formes reconnues par chaque grammaire choisie, tandis que le tableau 4.3 contient les statistiques des grammaires choisies. Un astérisque à la droite du nom d'une grammaire signifie que cette dernière est une grammaire pour la langue anglaise.

Tableau 4.2 Description des formes reconnues par les grammaires du corpus

	Formes reconnues
activate-now	Les dates d'activation immédiate d'un service
activation-date	Les dates d'activations d'un service
arrondissements	Les arrondissements de la ville de Montréal
attendant	Les noms du personnel d'une entreprise
commands*	Les commandes d'une application vocale
confirmation	Les confirmations possibles d'une question
credit-card-expiration*	Les dates d'expiration d'une carte de crédit
nas-dernier-triplet	Les derniers triplets d'un numéro d'assurance sociale canadien
postal-code	Les codes postaux canadiens
zip-code*	Les codes postaux américains

Tableau 4.3 Statistiques des grammaires du corpus

	Terminaux	Non-terminaux	Productions	Anullables
activate-now	14	11	25	4
activation-date	58	11	77	5
arrondissements	142	68	208	21
attendant	73	13	78	2
commands*	52	35	88	19
confirmation	18	11	31	3
credit-card-expiration*	58	27	92	13
nas-dernier-triplet	161	35	236	11
postal-code	252	84	376	31
zip-code*	28	15	40	7

Outre les mesures de similitudes, nous avons également pris soin d'accumuler d'autres statistiques lors de la phase d'évaluation des performances. Ces statistiques sont utiles principalement pour évaluer la performance en terme de rapidité plutôt qu'en terme de qualité des corrections. Le tableau 4.4 donne la liste des statistiques récoltées ainsi que leur signification. Ces statistiques sont prises individuellement pour chaque paire d'algorithme et de phrases possibles.

Tableau 4.4 Statistiques récoltées pour chaque phrases lors de l'évaluation des algorithmes

	Description
Distance de Levenshtein	Distance de Levenshtein entre la correction référence et la correction proposée
Distance de Jaccard	Distance de Jaccard entre la correction référence et la correction proposée
Temps écoulé initialisation (en ms)	Temps écoulé pour la phase d'initialisation de l'algorithme
Temps écoulé reconnaissance (en ms)	Temps écoulé pour la phase de reconnaissance de l'algorithme
Temps écoulé construction (en ms)	Temps écoulé pour la phase de construction de l'arbre syntaxique de l'algorithme
Nombre d'ensembles d'Earley	Nombre d'ensembles d'Earley produit pour effectuer la reconnaissance
Nombre d'états d'Earley	Nombre d'états d'Earley produit pour effectuer la reconnaissance

Les mesures de temps sont prises en millisecondes. Elles dénotent trois phases d'exécution différentes de chaque algorithme : la phase d'initialisation, de reconnaissance et de construction. La phase d'initialisation comporte toutes les opérations effectuées avant de mettre l'algorithme en marche pour une grammaire donnée. La phase de reconnaissance consiste en toutes les étapes requises pour peupler les différents ensembles d'Earley. Finalement, la phase de construction s'occupe quant à elle de parcourir les ensembles d'Earley afin de reconstruire la phrase corrigée.

Les deux autres statistiques sont issues des variables internes des algorithmes. Le nombre d'ensembles d'Earley correspond au nombre d'ensembles créés par l'algorithme pour faire l'analyse de la phrase. Le nombre d'états correspond au nombre d'états totaux créés par l'algorithme. Cette mesure est calculée en additionnant le nombre d'éléments dans chaque ensemble d'Earley.

À ce stade-ci de la section, nous avons enfin une vue d'ensemble sur la méthodologie qui sera utilisée pour comparer les algorithmes entre eux. Nous avons en effet deux métriques pour comparer des corrections entre elles ainsi qu'un corpus de corrections de références comportant 250 entrées réparties en part égale sur 10 grammaires. Cependant, nous voulons également prouver que les algorithmes d'analyse syntaxique robuste ne font pas de la sur-

correction. La surcorrection survient dans les cas où un algorithme détecterait une erreur de syntaxe dans une phrase qui est analysable par un algorithme classique, ce qui implique que la phrase ne contient pas d'erreurs. Ceci est important, car on veut s'assurer que les corrections données pour des phrases contenant des erreurs ne sont pas dues à un mauvais fonctionnement de l'algorithme.

Pour prouver ce point, nous avons développé un deuxième corpus de phrases, où toutes les phrases ne contiennent aucune erreur. Pour créer ce corpus de validation, nous avons utilisé l'outil *NuGram IDE* développé par *Nu Echo*. Cet outil permet de générer des phrases valides pour une grammaire donnée en fonction de certains critères. Il est par exemple possible de générer toutes les phrases d'une grammaire, de générer les phrases de façon à couvrir toutes les branches de la grammaire ou de générer les phrases de façon aléatoire. Nous avons décidé qu'il y aurait au maximum 100 phrases pour chaque grammaire choisie pour le corpus de comparaison. Pour chaque grammaire, nous avons demandé à l'outil de nous générer 100 phrases d'une manière aléatoire. Dans les cas où le langage généré par une grammaire contenait moins de 100 phrases, nous avons plutôt choisi de générer toutes les phrases de la grammaire en question.

Il est ensuite très aisé de valider que les algorithmes n'effectuent pas de surcorrection. Puisque qu'il n'y a pas d'erreurs dans les phrases, la correction pour n'importe quelle phrase du corpus est la phrase elle-même. Comme la phrase et la correction sont identiques, les deux métriques choisies pour la comparaison retourneront toutes deux 1.0 comme valeur. Nous avons établi qu'un algorithme ne faisait probablement aucune surcorrection de phrases s'il obtenait des métriques de 1.0 pour toutes les phrases contenues dans le corpus de validation.

Avant de terminer, nous désirons pointer le lecteur vers l'annexe B. Cette annexe contient les directives exactes qui ont été données aux personnes effectuant l'évaluation. On y parle notamment de la manière dont les phrases doivent être corrigées et des règles à suivre pour déterminer la correction de référence.

CHAPITRE 5

RÉSULTATS

Ce chapitre est consacré exclusivement aux résultats expérimentaux obtenus et à leurs analyses. Nous présentons dans cette section des tableaux et des graphiques qui couvrent l'ensemble de ces résultats. Les résultats bruts qui ont été utilisés pour produire tous les graphiques et les tableaux peuvent être consultés à l'annexe C.

Au fur et à mesure que les tableaux et graphiques sont présentés, nous faisons une analyse détaillée des éléments importants qui ressortent des résultats. On y apprend par exemple que l'algorithme d'Anderson-Backhouse est sensiblement équivalent à l'algorithme de Lyon en terme de qualité des corrections proposées (un écart moyen de 5.51% en faveur de l'algorithme de Lyon). Que l'algorithme d'Anderson-Backhouse est le plus rapide des trois étant 229% par rapport au deuxième plus rapide. Également, on note que l'algorithme hybride est sensiblement identique à l'algorithme de Lyon en terme de qualité de correction, mais en étant environ 2 fois plus rapide.

Nous désirons tout d'abord, à des fins de reproductibilité, présenter les caractéristiques de la machine ayant servi à collecter les résultats. Cette machine possédait un processeur *Intel Core 2 Duo E7400* cadencé à $2.79GHz$ ainsi que $2Go$ de mémoire vive. Le système d'exploitation de la machine était *Windows 7 Professional* 32 bits (version 6.1.7601). Les algorithmes étant implémentés en *Python*, c'est la version 3.2.1 de *Python* qui a été utilisée pour obtenir les résultats.

Ne perdons plus de temps et allons tout de suite au coeur de ce qui nous intéresse : quel algorithme est le plus performant en terme de qualité de correction ? Pour répondre à cette question, nous avons produit le graphique 5.1. Pour le produire, nous avons tout d'abord calculé la moyenne des résultats des grammaires pour chaque algorithme et chaque évaluateur. Avec ces nouvelles données, nous avons ensuite refait une moyenne des résultats des évaluateurs pour chaque algorithme.

La première chose que l'on peut dire sur ce graphique est que l'algorithme de correction globale est celui qui offre la meilleure correction. En effet, au niveau de la distance de Jaccard, cet algorithme est respectivement 3.84% et 7.93% meilleur que l'algorithme hybride et l'algorithme de correction locale. Au niveau de la distance de Levensthein, c'est plutôt 0.98% et 3.08%. Le tableau 5.1 récapitule le pourcentage de différence entre les algorithmes où l'algorithme de correction globale est la valeur étalon.

Malgré le fait que l'algorithme de correction globale offre les meilleures corrections, la

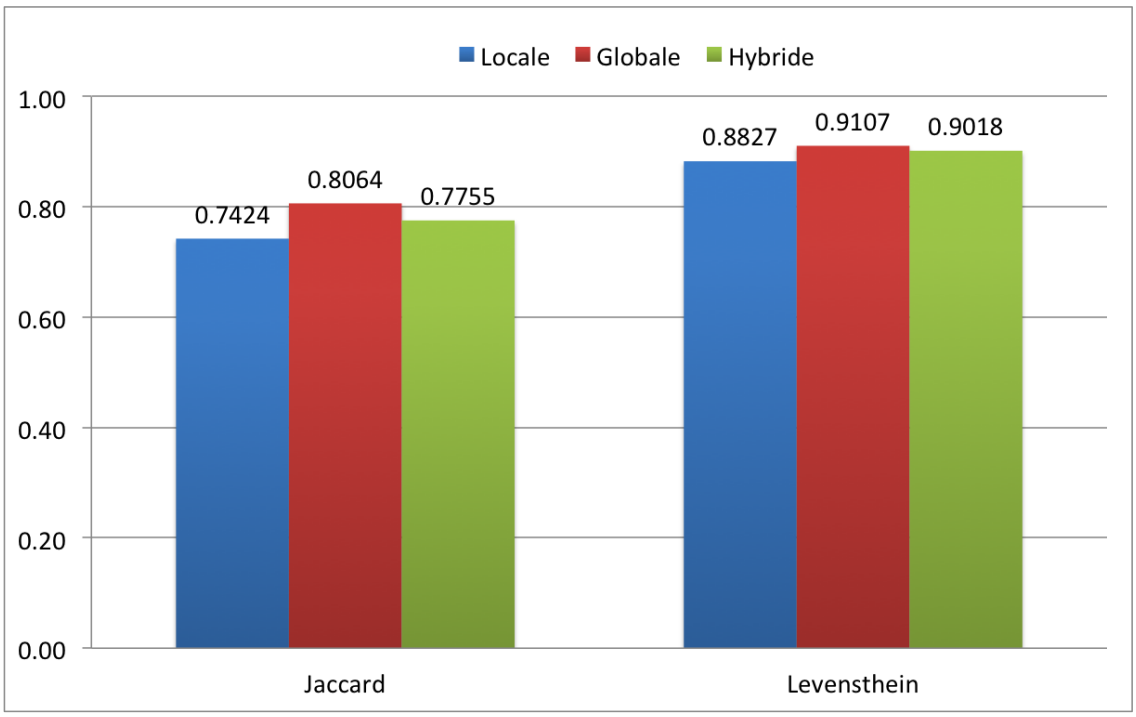

Figure 5.1 Comparaison des algorithmes - mesures de similitude

Tableau 5.1 Difference, en pourcentage, entre les algorithmes

	Jaccard	Levensthein
Locale	-7.93 %	-3.08 %
Globale	0.00 %	0.00 %
Hybride	-3.84 %	-0.98 %

différence entre eux n'est pas très importante. Ceci est quelque chose qui nous étonne quelque peu, car nous nous attendions à ce que ce dernier soit bien meilleur en comparaison des deux autres.

Mais avant d'analyser un autre élément de comparaison, nous désirons démontrer l'accord inter-évaluateurs. En effet, les données de mesure de similitude présentées ci-dessus ont été obtenues en faisant la moyenne des résultats de chaque évaluateur. Il est donc utile de connaitre à quel point les phrases choisies par les différents évaluateurs étaient près les unes des autres.

L'écart moyen des mesures de similitudes par rapport à la moyenne est une mesure intéressante pour estimer l'accord inter-évaluateurs, car pour calculer la similitude, seule la phrase fournie par l'évaluateur change. En effet, les grammaires restent constantes pour tous les évaluateurs tout comme la phrase déterminée comme la meilleure par les algorithmes. C'est

donc dire que l'on compare la meilleure phrase fournie par l'algorithme aux trois phrases potentiellement différentes que les évaluateurs ont déterminés comme la correction référence. Donc, la seule chose qui fait varier les mesures de similitude pour une phrase est ce que l'évaluateur a choisi. Plus l'écart moyen est petit, plus les phrases choisies par les différents évaluateurs étaient près les unes des autres.

Le tableau 5.2 présente cet écart moyen. Pour le calculer, nous avons pris la moyenne des résultats pour chaque grammaire. Nous avions donc des résultats pour 3 évaluateurs pour tous les algorithmes. L'écart moyen a été calculé à partir des résultats pour l'ensemble des évaluateurs. La moyenne pour chaque mesure, Levensthein et Jaccard, a été calculée puis nous avons ensuite fait la moyenne des écarts absolus pour chaque mesure de similitude par rapport à la moyenne.

Tableau 5.2 Écart moyen des mesures de similitude entre évaluateurs par algorithme

	Jaccard	Levensthein
Locale	0.0163	0.0083
Globale	0.0221	0.0131
Hybride	0.0178	0.0120

Quand on analyse le tableau 5.2, on voit rapidement que l'écart moyen est relativement petit. Le plus grand écart moyen est de 0.0221. Pour la distance de Jaccard, la moyenne des écarts moyens est de 0.0187 et de 0.0111 pour la distance de Levensthein. On peut donc affirmer que les évaluateurs étaient relativement d'accord sur la meilleure phrase à choisir dans la majorité des cas.

En plus de l'écart moyen des mesures de similitude, nous avons également comparé entre elles les différentes phrases qui ont été fournies par les évaluateurs. Pour chaque phrase de chaque grammaire, il y avait trois paires de phrases possibles à comparer : évaluateur #1 et évaluateur #2, évaluateur #1 et évaluateur #3 et finalement évaluateur #2 et évaluateur #3. Avec 25 phrases par grammaire et 10 grammaires, il y a donc 750 paires possibles. Pour l'ensemble des phrases, nous avons déterminé qu'il y avait 80.27 % de paires complètement identiques. Ces résultats nous démontrent que les évaluateurs étaient relativement d'accord sur la meilleure correction à apporter à la phrase.

Pour terminer l'analyse des mesures de similitudes, nous désirons également présenter les figures 5.2 et 5.3. Ces deux figures montrent une moyenne des résultats des évaluateurs, groupés par grammaire et par algorithme.

Ces deux graphiques illustrent le fait que selon la grammaire, l'algorithme fournissant la meilleure correction peut être différent. Pour la grammaire *activate-now*, c'est l'algorithme de

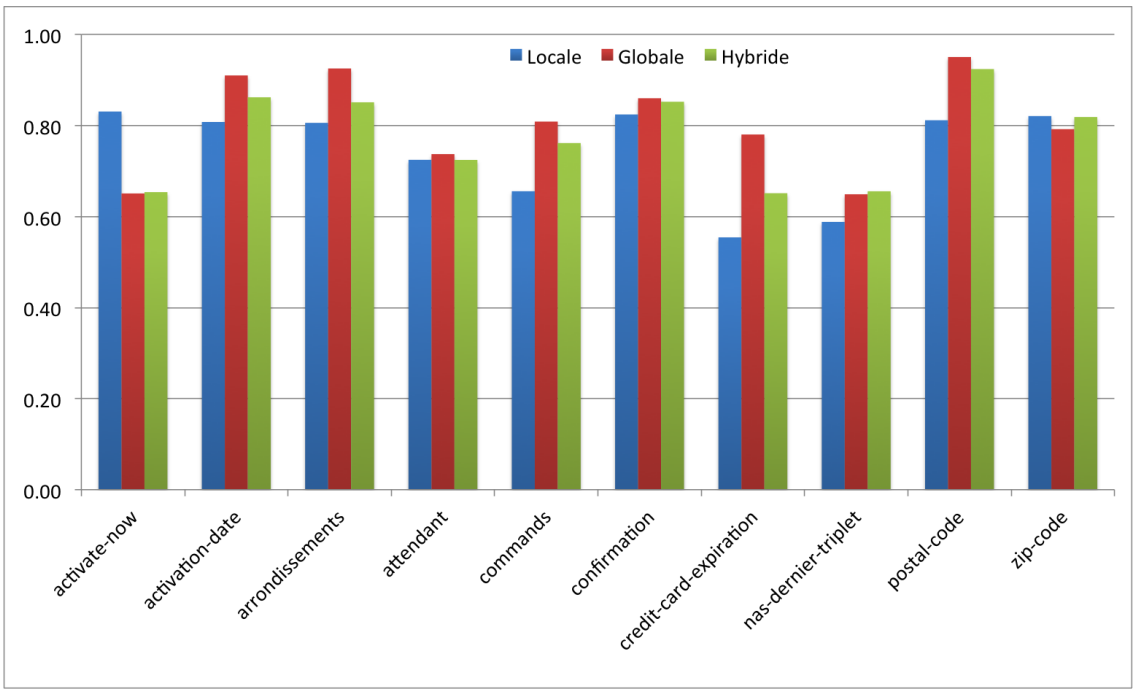

Figure 5.2 Comparaison des algorithmes par grammaire - distance de Jaccard

correction locale qui offre la meilleure correction des phrases. Tandis que pour la grammaire *zip-code*, c'est plutôt l'algorithme hybride qui sort gagnant. Malgé tout, dans la majorité des grammaires, le meilleur est l'algorithme de Lyon, toujours suivi de près par l'algorithme hybride.

Puisque tout les algorithmes sont relativement près l'un de l'autre au niveau de la qualité des corrections, d'autres éléments sont nécessaires afin de mieux départager les algorithmes entre eux.

Passons donc maintenant à notre deuxième élément de comparaison : le temps écoulé. Trois mesures distinctes ont été prises et représentent chacune une phase particulière des algorithmes. Il y a le temps d'initialisation, le temps pour analyser la phrase et le temps pour construire une phrase corrigée à partir des items qui se trouvent dans les ensembles d'Earley. La première mesure est fixe, peu importe le nombre de phrases que l'algorithme doit analyser pour une grammaire donnée. Donc, si l'algorithme prend i millisecondes d'initialisation, a millisecondes d'analyse et c millisecondes de construction, le temps total écoulé pour n phrases sera $t = i + n * (a + c)$.

Le tableau 5.3 présente les mesures obtenues au niveau du temps écoulé. Ces mesures ont été calculées d'une manière similaire à celle des mesures de similitudes, c'est-à-dire que ce sont les moyennes des grammaires puis des évaluateurs. La dernière colonne du tableau représente le total du temps écoulé par phrase, soit la somme du temps de reconnaissance et

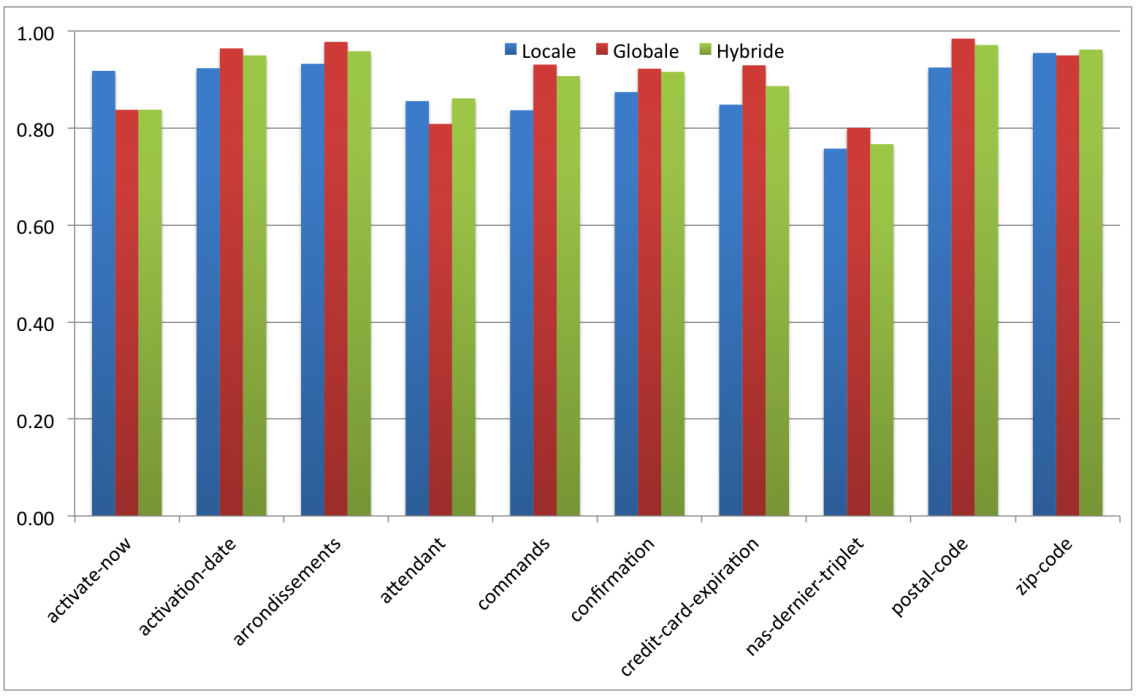

Figure 5.3 Comparaison des algorithmes par grammaire - distance de Levensthein

Tableau 5.3 Temps moyen écoulé (ms) par algorithme

	Initialisation	Reconnaissance	Construction	Total (par phrase)
Locale	153.30	542.22	0.68	542.90
Globale	0.65	2198.19	39.49	2237.68
Hybride	0.59	1242.91	18.63	1243.81

du temps de construction.

Attardons-nous en premier au temps écoulé total par phrase. Dans cette colonne, on remarque rapidement que l'algorithme le plus rapide, en moyenne, est l'algorithme de correction locale des erreurs. Ce dernier est 229 % plus rapide que son plus proche poursuivant, l'algorithme hybride. Si on le compare plutôt à l'algorithme de correction globale, le plus lent, il est 412 % plus rapide.

Au niveau du temps d'initialisation, l'algorithme de correction locale est toutefois moins rapide, beaucoup moins même. Ceci a toutefois un impact limité puisque le temps d'initialisation est calculé une seule fois par grammaire.

Cette conclusion au niveau du temps écoulé était assez prévisible à cause de la manière dont les algorithmes fonctionnent. En effet, l'algorithme de correction globale, et l'hybride à un certain point, évaluent beaucoup plus de possibilités que l'algorithme de correction locale.

Plus il y a de possibilités, plus le temps écoulé sera important. Les résultats à ce niveau ne sont donc pas une surprise. Cependant, la différence entre les algorithmes est tout de même très importante. Pour une seule phrase, les temps sont respectables. Mais si on a à analyser une certaine quantité de phrases, le temps peut devenir rapidement trop long.

Cependant, il faut relativiser quelque peu les données du temps écoulé. Le temps écoulé est proportionnel à deux facteurs : la taille de la grammaire en terme de productions, de terminaux et de non-terminaux ainsi que le nombre de mots dans la phrase. Nous avons pris 10 grammaires qui étaient différentes les unes des autres particulièrement en ce qui concerne la taille.

Tableau 5.4 Écart moyen du temps écoulé (ms) entre grammaires par algorithme

	Initialisation	Reconnaissance	Construction	Total (par phrase)
Locale	267.02	861.74	0.69	862.14
Globale	0.85	5156.85	96.39	5173.32
Hybride	0.79	2745.76	39.99	2753.34

Si on regarde le tableau 5.4, on remarque rapidement que l'écart moyen par rapport à la moyenne de tous les algorithmes est très élevé. Ceci se produit à cause de la variabilité dans la taille des grammaires. Mais encore une fois, c'est l'algorithme de correction locale qui s'en sort le mieux et c'est l'algorithme de correction globale qui est le pire.

Pour les grammaires relativement petites, la différence entre les algorithmes est assez faible en terme de temps écoulé. Le tableau 5.5 reprend les mesures du temps écoulé pour la grammaire la plus petite du lot, la grammaire *activate-now* (14 terminaux, 11 non-terminaux et 25 productions). On peut voir que le temps écoulé est relativement court pour tous les algorithmes. Même si l'algorithme de correction locale reste le plus rapide, la différence absolue entre les algorithmes (environ 15 ms) n'est pas vraiment perceptible du point de vue d'un utilisateur.

Tableau 5.5 Temps écoulé (ms) pour la grammaire *activate-now* par algorithmes

	Initialisation	Reconnaissance	Construction	Total (par phrase)
Locale	1.47	14.68	0.35	15.03
Globale	0.08	28.46	0.62	29.08
Hybride	0.07	26.94	0.63	27.58

Cependant, pour les grammaires de grande taille, c'est une autre histoire. Les algorithmes de correction globale et hybride ne sont pas capables de travailler dans des temps raisonnables.

Pour démontrer ce problème, regardons le tableau 5.6 qui montre les mesures du temps écoulé pour la plus grosse grammaire du corpus, la grammaire *postal-code* (252 terminaux, 84 non-terminaux et 376 productions).

Tableau 5.6 Temps écoulé (ms) pour la grammaire *postal-code* par algorithmes

	Initialisation	Reconnaissance	Construction	Total (par phrase)
Locale	677.50	1774.99	2.55	1777.54
Globale	2.74	16709.00	48.60	16757.60
Hybride	2.55	8921.06	20.10	8941.16

Quand on regarde ce tableau, la valeur la plus élevée est le temps total écoulé par phrase pour l'algorithme de correction globale qui est en moyenne de 16757.6 millisecondes, soit environ 17 secondes. Ceci est vraiment très élevé même pour faire l'analyse d'une seule phrase d'une manière très précise. Cela devient encore pire quand il faut analyser un ensemble de phrases. Par exemple, pour analyser 1000 phrases avec la grammaire *postal-code*, l'algorithme de correction globale prendra environ 4 heures 40 minutes et l'algorithme hybride environ 2 heures 48 minutes. Pour le même nombre de phrases, l'algorithme de correction locale fera le travail en seulement 30 minutes.

CHAPITRE 6

CONCLUSION

Ce chapitre est dédié aux conclusions que nous avons tirées des données récoltées lors de la phase d'expérimentation. Au début, nous présentons une synthèse rapide des travaux et des conclusions qui en découlent. Nous terminons avec une liste des améliorations envisageables et une description des travaux futurs.

6.1 Synthèse des travaux

Dans le cadre de ce projet de recherche, nous avons cherché à savoir si un algorithme d'analyse syntaxique avec correction locale des erreurs peut rivaliser, en terme de qualité de correction, avec un algorithme effectuant une correction globale des erreurs.

Pour effectuer cette comparaison, nous avons implémenté trois algorithmes d'analyse syntaxique robuste. Tout d'abord, l'algorithme d'Anderson-Backhouse qui représente l'algorithme effectuant une correction locale des erreurs. Ensuite, nous avons décidé d'utiliser l'algorithme de Lyon comme algorithme de correction globale de référence. Finalement, nous avons également développé un troisième algorithme que nous avons appelé l'algorithme hybride, afin d'améliorer la performance en terme de temps de l'algorithme de Lyon.

La qualité d'une correction particulière étant un concept assez abstrait, nous avons élaboré une méthodologie particulière afin de comparer les algorithmes entre eux. Avec cette méthodologie, nous sommes en mesure de modéliser adéquatement le concept de qualité de correction. Grâce à cette modélisation, nous pouvons comparer deux corrections entre elles et quantifier leur similitude.

À l'aide d'un corpus de grammaires et de phrases provenant de cas d'utilisations réels, nous avons obtenu une multitude de résultats expérimentaux. Nous nous sommes ensuite attelés à la tâche d'analyser toutes ces données.

Du côté de la qualité des corrections, nous pouvons affirmer que l'algorithme de Lyon est effectivement l'algorithme offrant les meilleures corrections. Cependant, l'écart avec les deux autres algorithmes est plutôt faible n'étant en moyenne que de 2.41 % avec l'algorithme hybride et de 5.51 % avec l'algorithme d'Anderson-Backhouse.

Au niveau du temps écoulé, c'est l'algorithme d'Anderson-Backhouse qui est le plus rapide, et de loin. Avec notre corpus, l'algorithme prend en moyenne 542.90 ms pour analyser une phrase, l'algorithme hybride, 1243.81 ms et l'algorithme de Lyon 2237.68 ms. C'est donc dire

que l'algorithme d'Anderson-Backhouse est environ 2 et 4 fois plus rapide que l'algorithme hybride et l'algorithme de Lyon respectivement.

En étant 5.51 % moins performant que l'algorithme avec correction globale, l'algorithme de correction locale offre une alternative très intéressante en terme de qualité des corrections. Il devient encore plus intéressant si on tient compte du temps écoulé puisqu'il est beaucoup plus rapide que les autres algorithmes.

Avant de terminer, nous aimerions mentionner la bonne prestation de l'algorithme hybride. En étant presqu'équivalent à l'algorithme de Lyon en terme de qualité de correction, il permet un gain d'environ 50 % au niveau du temps écoulé par rapport à ce dernier.

6.2 Améliorations et travaux futurs

Différentes avenues sont envisageables pour améliorer les algorithmes que nous avons implémentés.

Pour l'algorithme de Lyon, nous pensons être en mesure d'améliorer son efficacité en utilisant une version plus rapide de l'algorithme d'Earley. En effet, l'idée générale qui a permis de rendre l'algorithme d'Earley robuste aux erreurs pourrait être réutilisée pour être appliquée à un algorithme similaire. L'algorithme de Leo (Leo, 1991) pourrait par exemple être utilisé puisque c'est une version améliorée de l'algorithme d'Earley qui s'exécute dans un temps linéaire pour toutes les grammaires LR(k).

Pour l'algorithme d'Anderson-Backhouse, nous croyons qu'il n'y a pas beaucoup d'améliorations possibles. Cependant, nous pensons qu'il est possible d'améliorer sensiblement l'algorithme hybride que nous avons développé.

L'objectif serait d'en faire un vrai algorithme de correction régionale. Pour y parvenir, il faudrait être en mesure de revenir au fonctionnement classique après avoir enclenché la correction des erreurs. Le défi est de décider à quel moment il devient opportun de retourner à une analyse classique.

Tous les algorithmes ont été implémentés en utilisant une base de travail commune. Par exemple, les structures de données utilisées pour bâtir les états ainsi que les ensembles sont les mêmes. En utilisant des structures plus efficaces et en implémentant les algorithmes dans un langage compilé, nous croyons que nous pourrions améliorer significativement le temps écoulé.

Dès le tout début, le projet de recherche a toujours été une prémisse au développement d'outils pouvant améliorer des grammaires hors contexte. À ce niveau, nous envisageons le développement de deux outils.

Le premier serait un outil interactif qui utiliserait l'algorithme d'Anderson-Backhouse afin

de montrer graphiquement où se trouvent les erreurs. L'idée est de bâtir un arbre syntaxique d'une phrase où les erreurs seraient affichées de différentes couleurs. Un mot supprimé pourrait être colorié en rouge par exemple. Cet outil permettrait au développeur de la grammaire de voir immédiatement pourquoi une phrase n'a pu être analysée.

Le deuxième, plus ambitieux cette fois, serait un outil complètement automatique. Ce que nous envisageons est d'utiliser l'algorithme d'Anderson-Backhouse afin d'analyser un ensemble très grand de phrases. À l'aide des diagnostics fournis par l'algorithme pour chaque phrase, le but de l'outil serait de déterminer les patrons d'erreurs les plus fréquents. À partir de ces patrons, nous serions alors en mesure d'inférer des améliorations à apporter à la grammaire afin d'en augmenter sa couverture. La difficulté réside dans la dualité spécialisation-généralisation : proposer des améliorations qui augmentent la couverture tout en gardant la grammaire aussi spécifique que possible.

Avec cette mutlitude de travaux futurs, cette recherche n'est que la pointe de l'iceberg et une foule de travail reste à faire dans notre quête d'outils de diagnostics des erreurs de syntaxe.

RÉFÉRENCES

AHO, A. V. et PETERSON, T. G. (1972). A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1, 305–312.

ANDERSON, S. O. et BACKHOUSE, R. C. (1981). Locally least-cost error recovery in earley's algorithm. *ACM Trans. Program. Lang. Syst.*, 3, 318–347.

BLACK, N. et MOORE, S. (1994). Gauss-seidel method, (dernier accès : 7 mars 2013). `http://mathworld.wolfram.com/Gauss-SeidelMethod.html`.

BRUSCHI, D. et PIGHIZZINI, G. (1992). A parallel general context-free parser.

COCKE, J. (1969). *Programming languages and their compilers : Preliminary notes*. Courant Institute of Mathematical Sciences, New York University.

DION, B. et FISCHER, C. (1978). A least-cost error corrector for lr (1)-based parsers. Rapport technique, Tech. Rep. 333, Univ. of Wisconsin, Madison, Sept.

EARLEY, J. (1970). An efficient context-free parsing algorithm. *Commun. ACM*, 13, 94–102.

GRUNE, D. et JACOBS, C. J. H. (1990). *Parsing Techniques : A Practical Guide*. Computers and their Applications. Ellis Horwood, Chichester, UK.

HEINECKE, J., KUNZE, J., MENZEL, W. et SCHRÖDER, I. (1998). Eliminative parsing with graded constraints. *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*. Association for Computational Linguistics, Stroudsburg, PA, USA, ACL '98, 526–530.

IRONS, E. (1963). An error-correcting parse algorithm. *Communications of the ACM*, 6, 669–673.

KARLSSON, F. (1990). Constraint grammar as a framework for parsing running text. *Proceedings of the 13th conference on Computational linguistics - Volume 3*. Association for Computational Linguistics, Stroudsburg, PA, USA, COLING '90, 168–173.

KASAMI, T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Rapport technique, Air Force Cambridge Research Lab, Bedford, MA.

KNUTH, D. E. (1965). On the translation of languages from left to right. *Information and Control*, 8, 607–639.

KURODA, K.-I. et TANAKA, E. (1993). An error-correcting version of the leiss's parser for context-free languages. *IEICE TRANSACTIONS on Information and Systems*, 76, 1528–1531.

LEE, K. J., KWEON, C. J., SEO, J. et KIM, G. C. (1995). A robust parser based on syntactic information. *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, EACL '95, 223–228.

LEISS, H. (1990). On kilbury's modification of earley's algorithm. *ACM Trans. Program. Lang. Syst.*, 12, 610–640.

LEO, J. M. I. M. (1991). A general context-free parsing algorithm running in linear time on every lr (k) grammar without using lookahead. *Theor. Comput. Sci.*, 82, 165–176.

LEVENSHTEIN, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10, 707–710.

LÉVY, J. P. (1975). Automatic correction of syntax-errors in programming languages. *Acta Informatica*, 4, 271–292. 10.1007/BF00288730.

LYON, G. (1974). Syntax-directed least-errors analysis for context-free languages : a practical approach. *Commun. ACM*, 17, 3–14.

MAUNEY, J. et FISCHER, C. N. (1982). A forward move algorithm for ll and lr parsers. *SIGPLAN Not.*, 17, 79–87.

MAUNEY, J. et FISCHER, C. N. (1988). Determining the extent of lookahead in syntactic error repair. *ACM Trans. Program. Lang. Syst.*, 10, 456–469.

MELLISH, C. (1989). Some chart-based techniques for parsing ill-formed input. *Proceedings of the 27th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 102–109.

PIGHIZZINI, G. (1992). A parallel minimum distance error-correcting context-free parser.

ROSENKRANTZ, D. J. et STEARNS, R. E. (1969). Properties of deterministic top down grammars. *Proceedings of the first annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, STOC '69, 165–180.

TAI, K. (1978). Syntactic error correction in programming languages. *Software Engineering, IEEE Transactions on*, 414–425.

TOMITA, M. (1985). An efficient context-free parsing algorithm for natural languages. *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 756–764.

VANRULLEN, T., BLACHE, P. et BALFOURIER, J.-M. (2006). Constraint-Based Parsing as an Efficient Solution : Results from the Parsing Evaluation Campaign EASy. *Proceedings of LREC 2006 (Language Resources and Evaluation)*. LREC, 165–170. 2771.

VILARES, M., DARRIBA, V. M. et RIBADAS, F. J. (2000). Regional least-cost error repair. *Implementation and Application of Automata, volume 2088 of LNCS*. Springer-Verlag, 293–301.

WAGNER, R. A. et FISCHER, M. J. (1974). The string-to-string correction problem. *J. ACM*, 21, 168–173.

WEISCHEDEL, R. M. et SONDHEIMER, N. K. (1983). Meta-rules as a basis for processing ill-formed input. *Comput. Linguist.*, 9, 161–177.

WIKIPEDIA (2005). Jaccard index, (dernier accès : 7 mars 2013). `http://en.wikipedia.org/wiki/Jaccard_index`.

YOUNGER, D. H. (1966). Context-free language processing in time n3. *Switching and Automata Theory, 1966., IEEE Conference Record of Seventh Annual Symposium on*. 7 –20.

ANNEXE A

ALGORITHME D'ANDERSON-BACKHOUSE - DÉTAILS SUPPLÉMENTAIRES

Cette annexe contient les détails précis de certaines fonctions utilisées par l'algorithme d'Anderson-Backhouse. On y aborde les détails des fonctions *completionCost*, *computeCompletionCosts*, *predictCompletionCosts* ainsi que du vecteur *completionCosts*.

Rappelons-nous que la fonction *findBestEditOperation* effectue une itération sur tous les items de la forme $[A \rightarrow \alpha \bullet \mathbf{t}\ \beta, f, j]$ dans l'ensemble s_j. Deux calculs sont effectués pour déterminer la meilleure opération d'édition. Le coût d'incomplétion est le premier tandis que le coût d'insertion plus le coût de complétion est le deuxième. C'est ce dernier coût que nous cherchons à détailler. Le deuxième calcul est obtenu comme suit : $cost^*(\epsilon, \mathbf{t}\ \beta) + completionCosts[j][A, t_i]$.

Décortiquons quelque peu cette dernière équation. La première partie, $cost^*(\epsilon, \mathbf{t}\ \beta)$, calcule le coût d'insérer la séquence $\mathbf{t}\ \beta$. Le fait d'insérer la séquence permet de complèter l'item.

La deuxième partie, $completionCost[j][A, t_i]$, est un tableau à deux dimensions indexé par j puis par la paire A, t_i. Ce tableau est peuplé par la fonction *predictCompletionCosts*, décrite un peu plus loin dans cette annexe.

Le coût de complétion établit le coût minimal pour transformer t_i par un des préfixes suivant le non-terminal A dans la règle de départ Z. Une autre manière de voir le coût de complétion est de penser que c'est le coût de continuer l'analyse suivant le non-terminal A sachant qu'on traite le mot original t_i et qu'il y a présentement j mots dans la phrase corrigée

Pour mieux comprendre cette idée, nous allons définir la fonction
$openPortion(A, j)$ (A.1) qui représente les préfixes de la partie ouverte de A. La partie ouverte représente ce qui peut suivre A dans la règle de départ Z.

$$openPortion(A, j) = \left\{ v \mid \forall vw \wedge Z \underset{G}{\Rightarrow}^* u_0, u_1, \ldots, u_{j-1} Avw \wedge v \in T^+ \wedge w \in T^* \right\} \qquad \text{(A.1)}$$

Nous allons maintenant introduire la fonction *completionCost* (A.2). Cependant, sa définition est telle qu'il n'est pas possible de la convertir en un algorithme utilisable par un ordinateur. Ces difficultés sont énumérées et contournées dans les paragraphes qui suivent.

Même si la fonction *completionCost* n'est pas utilisée dans notre projet, le fait de l'expliquer permet toutefois de mieux comprendre la manière dont est calculé le vecteur des coûts

de complétion qui est décrit un peu plus loin.

$$\textit{completionCost}(j, A, t_i) = \textit{incompletionCost*}(t_i, \textit{openPortion}(A, j)) \tag{A.2}$$

Quand on regarde la définition A.2, on voit que le coût de complétion équivaut à calculer le coût d'incomplétion pour transformer t_i par une des séquences de la partie ouverte de A suivant j. En tenant compte de la définition de la fonction *incompletionCost**, cela équivaut à insérer les $m-1$ premiers symboles d'une des séquences puis d'accepter ou changer t_i par le dernier symbole de la séquence. Prenons un exemple concret pour bien comprendre les notions précédentes.

L'algorithme cherche la meilleure correction pour le terminal $t_i = \mathbf{d}$ faisant partie de la phrase initiale non corrigée **a b d e**. L'item présentement analysé dans la boucle de la fonction *findBestEditOperation* est $[C \rightarrow \bullet\ \mathbf{c}, 2, 2]$. Assumons également que la phrase corrigée est **a b** et que la grammaire G contient les productions $Z \rightarrow \mathbf{a\ b}\ C\ \mathbf{d\ e}$ et $C \rightarrow \mathbf{c}$.

On cherche à calculer le coût pour corriger t_i par ce qui peut suivre C lorsque **a b** a déjà été corrigée. Ce qui peut suivre C après avoir corrigé 2 mots est la partie ouverte de C avec $j = 2$ qui est donnée par $\textit{openPortion}(C, 2)$. Dans notre exemple, $\textit{openPortion}(C, 2) = \{\mathbf{d}, \mathbf{d\ e}\}$ puisque ce sont les deux préfixes pouvant suivre le non-terminal C lorsque **a b** se retrouve dans la liste de mots corrigée. Le coût de complétion $\textit{completionCost}(2, C, \mathbf{d})$ est alors obtenu en appelant $\textit{incompletionCost*}(\mathbf{d}, \{\mathbf{d}, \mathbf{d\ e}\})$.

La définition purement mathématique *openPortion* (A.1) dans la fonction *completionCost* pose un problème au niveau de l'implémentation. En effet, la fonction *openPortion* génère tous les préfixes de terminaux pouvant suivre le non-terminal A. Ceci est un obstacle, car il est impossible de générer toutes les possibilités de séquences pour certaines grammaires, par exemple les grammaires récursives. Pour être en mesure de calculer les coûts de complétion, il faut éviter la génération de toutes les séquences de terminaux possibles pouvant suivre A.

La première chose à faire est de changer quelque peu la définition de $\textit{openPortion}(A, j)$. Au lieu de retourner des séquences de terminaux, la fonction est modifiée afin de plutôt retourner des séquences de symboles ($v \in (N \cup T)^+$). L'idée derrière ce changement est d'éviter la génération des terminaux pouvant être produits par les non-terminaux. Comme les résultats de $\textit{openPortion}(A, j)$ sont utilisés exclusivement dans la fonction *incompletionCost**, cela ne pose pas de problèmes, car cette dernière est capable de traiter des non-terminaux.

La deuxième chose est de remarquer que la continuation de ce qui doit suivre A après avoir corrigé j mots est encodée directement dans les items d'Earley qui sont utilisés par l'algorithme. En effet, lorsque l'algorithme commence l'analyse du mot t_i, la phase de prédiction rencontre des items de la forme $[B \rightarrow \alpha \bullet A\ \beta, f, c]$. L'élément β qui se trouve après le non-terminal A dans l'item encode la continuation de A après avoir corrigé j mots. À l'aide

de la grammaire G, il est possible d'utiliser β afin de reconstruire, jusqu'à la règle de départ Z, toutes les séquences de symboles pouvant suivre A.

À l'aide de ces deux idées, il est possible de décrire l'algorithme utilisé par les auteurs pour parvenir à pallier au problème de la définition des préfixes de la partie ouverte. Les auteurs adoptent le concept de vecteur de coûts de complétion. Ce vecteur contient tous les coûts de complétion pour les différentes combinaisons de règles, de terminaux et de nombre de mots corrigés qui peuvent survenir durant toute la durée de l'algorithme. Ce vecteur est peuplé élément par élément durant chaque phase de prédiction de l'algorithme.

C'est la fonction *predictCompletionCosts*, dont nous avons parlé précédemment dans l'algorithme 3.2, qui s'occupe de peupler le vecteur de coûts de complétion.

La notation $\mathit{completionCosts}[j][A,t]$ est utilisée pour représenter le vecteur de coûts de complétion. L'élément $\mathit{completionCosts}[j][A,t]$ retourne le coût de remplacer t par un préfixe suivant A après avoir corrigé j mots.

Au départ, la fonction *predictCompletionCosts* itère sur tous les non-terminaux et pour chaque non-terminal, itère ensuite sur tous les terminaux de la grammaire et initialise $\mathit{completionCosts}[j][X,t]$ à la valeur ∞. De cette façon, toutes les valeurs du vecteur de coûts de complétion pour j sont à leur valeur maximale.

Après avoir initialisé le vecteur $\mathit{completionCosts}[j]$, la fonction boucle sur tous les items de s_j qui sont de la forme $[B \rightarrow \alpha \bullet A\ \beta, f, j]$. Ce sont les items bloqués à un non-terminal. Pour chaque item, la fonction parcourt ensuite tous les terminaux de la grammaire et utilise la formule A.3 pour calculer la valeur de $\mathit{completionCosts}[j][A,t]$.

$$\mathit{completionCosts}[j][A,t] = \min \begin{pmatrix} \mathit{incompletionCost}^*(t, \mathit{prefixes}(\beta)), \\ \mathit{cost}^*(\epsilon, \beta) + \mathit{completionCosts}[f][B,t], \\ \mathit{completionCosts}[j][A,t] \end{pmatrix} \tag{A.3}$$

Pour remédier au problème de cycles qui peuvent survenir avec les grammaires récursives, la fonction *computeCompletionCosts* utilise la technique de Gauss-Seidel (Black et Moore, 1994) pour remplir itérativement les valeurs des coûts de complétion. Cette technique est implémentée en englobant la formule A.3 dans une boucle. Cette boucle continue tant et aussi longtemps qu'un des coûts de complétion du vecteur a changé durant la dernière itération. De cette manière, les coûts de complétion étant initialement assignée à ∞ vont, à chaque itération, converger vers leurs valeurs finales. Quand un coût peut dépendre de la valeur d'un autre, celui-ci sera mis à jour lors d'une itération subséquente. On arrête la boucle lorsque les coûts se sont stabilisés, c'est-à-dire qu'ils n'ont pas changés durant l'itération précédente.

Cependant, si l'on regarde la définition A.1 de la portion ouverte suivant A et qu'on la

compare à la définition A.3 qui permet de calculer le coût de complétion, il y a une différence significative. En effet, la partie ouverte doit inclure tout ce qui peut suivre A et doit former un suffixe valide de Z. Ce fait n'est pas reflété dans la fonction $completionCosts[j][A,t]$, car cette dernière utilise les préfixes du β de l'item au lieu des préfixes de la partie ouverte. Si on utilise la définition A.3 telle quelle, cela posera problème, car certains suffixes suivant A ne seront pas pris en compte.

Par exemple, prenons les productions $Z \rightarrow \mathbf{a}\ T \dashv$, $T \rightarrow B\ \mathbf{c}$ et $B \rightarrow \mathbf{b}$ qui peuvent produire seulement la phrase $\mathbf{a}\ \mathbf{b}\ \mathbf{c}\ \dashv$ en considérant que Z est la règle de départ. Si on veut calculer $completionCosts[1][B, \dashv]$, on trouvera l'item $[T \rightarrow \bullet\ B\ \mathbf{c}, f, 1]$ dans l'ensemble s_1. Ce qui suit B dans cet item est $\mathbf{c}$ et c'est ce qui sera utilisé pour calculer le coût de complétion. Puisque $\mathbf{c}$ ne peut être remplacé par $\dashv$, la fonction retournera ∞ comme coût, ce qui empêchera toutes corrections si $\mathbf{c}$ n'est pas dans la phrase originale.

Si on utilise la définition exacte de A.1 avec l'exemple précédent, il faudrait que β soit $\mathbf{c}\ \dashv$, car c'est la seule chaîne pouvant suivre B dans Z après avoir reconnu 1 mot. Avec ce β, la fonction va additionner le coût d'insérer $\mathbf{c}$ au coût d'accepter $\dashv$ ce qui permettra une correction si $\mathbf{c}$ est manquant dans la phrase.

On ne peut donc pas utiliser directement le β de l'item $[B \rightarrow \alpha \bullet A\ \beta, f, j]$ car cela ne respecte pas la définition de la partie ouverte. Pour respecter la définition de la partie ouverte, nous introduisons le concept de suffixes racines. Les suffixes racines sont les séquences de terminaux et non-terminaux qui peuvent suivre A dans la règle racine, d'où le nom suffixes racines. Le calcul des suffixes racines se fait d'une manière récursive.

Reprenons les productions utilisées précédemment pour illustrer la manière de calculer les suffixes racines. Supposons que β est $\mathbf{b}$ et que l'on veut trouver les suffixes racines suivant B. L'idée est simplement de trouver la production de la grammaire G qui a générée B. Cette production est $T \rightarrow B\ \mathbf{c}$. On remplace l'occurrence de B dans cette production par le β actuel ce qui nous donne un nouveau suffixe $\mathbf{b}\ \mathbf{c}$. On répète cette opération encore, ce qui nous amène à la production qui a générée T soit $Z \rightarrow \mathbf{a}\ T \dashv$ et on remplace encore T par le suffixe actuel. Ceci nous amène au nouveau suffixe $\mathbf{b}\ \mathbf{c}\ \dashv$. Ce dernier est un suffixe racine car la règle qui a produit ce suffixe est la règle racine.

Pour éviter une récursion infinie, chaque fois qu'on remplace un suffixe dans un non-terminal parent, on garde en mémoire la production parente. On s'assure ensuite, quand on cherche les productions parentes, que si elle a déjà été visitée, on ne la sélectionne pas de nouveau. De plus, plusieurs productions peuvent avoir générées un non-termial particulier. Il faut alors garder une liste des suffixes et faire remonter chacun d'entre eux jusqu'à ce tous les suffixes soient des suffixes racines.

Nous avons maintenant tous les éléments nécessaires pour proposer les algorithmes des

fonctions *predictCompletionCosts* et *computeCompletionCosts*.

La fonction *predictCompletionCosts* (A.1) reçoit en entrée l'ensemble s_j. Au départ, la fonction initialise le vecteur de complétion pour j de manière à ce que toutes les cases soient à leurs valeurs maximales, soit ∞. À ce point, une boucle est lancée et continue tant et aussi longtemps que le sous-vecteur *completionCosts*$[j]$ a changé. Ceci permet de faire converger tous les coûts jusqu'à ce qu'ils se soient stabilisés.

Ensuite, la fonction boucle sur tous les items de la forme $[B \rightarrow \alpha \bullet A\ \beta, f, j]$ dans s_j. Ces items sont les items de prédictions, c'est-à-dire qu'ils sont bloqués à un non-terminal. Pour chacun d'entre eux, la fonction envoie l'item à la fonction *computeCompletionCosts* pour que cette dernière calcule le coût de complétion de *completionCosts*$[j][A, t]$.

Algorithme A.1 Fonction *predictCompletionCosts* de l'algorithme d'Anderson-Backhouse

Entrée(s) : s_j l'ensemble d'Earley utilisé pour la prédiction
  pour tout X **dans** N **faire**
    pour tout t **dans** T **faire**
      completionCosts$[j][X, t] = \infty$

  repéter
    previous $\leftarrow$ *completionCosts*$[j]$
    pour tout *item* **de la forme** $[B \rightarrow \alpha \bullet A\ \beta, f, j]$ **dans** s_j **faire**
      computeCompletionCosts(*item*)
  jusqu'à *completionCosts*$[j]$ = *previous*

La fonction *computeCompletionCosts* (A.2) utilise ensuite les différents éléments que nous avons décrits précédemment pour calculer le coût de complétion. Elle commence par déterminer tous les suffixes racines. Elle boucle ensuite sur chaque terminal de la grammaire et pour chaque terminal, boucle sur chaque suffixe racine puis calcul le coût de complétion d'un élément du vecteur.

Algorithme A.2 Fonction *computeCompletionCosts* de l'algorithme d'Anderson-Backhouse

Entrée(s) : *item* de la forme $[B \rightarrow \alpha \bullet A\ \beta, f, j]$
  pour tout t **dans** T **faire**
    pour tout *suffix* **dans** *computeRootSuffixes*(β) **faire**

$$completionCosts[j][A,t] \leftarrow \min \begin{pmatrix} incompletionCost^*(t, suffix), \\ cost^*(\epsilon, suffix) + completionCosts[f][B,t], \\ completionCosts[j][A,t] \end{pmatrix}$$

Grâce à la fonction *computeCompletionCosts* il est possible d'obtenir, après chaque phase de prédiction, le vecteur des coûts de complétion. C'est ce vecteur qui est ensuite utilisé dans la fonction *findBestEditOperation* afin de déterminer la meilleure opération d'édition à appliquer au mot t_i.

ANNEXE B

DIRECTIVES AUX CORRECTEURS

Cette annexe contient les directives qui ont été données aux personnes devant fournir les corrections de référence. Elles ont été données en même temps que le corpus de phrases et les grammaires.

L'archive que vous avez reçu contient 10 fichiers de résultats que vous devez remplir, 10 fichiers de grammaires au format *Augmented Backus–Naur Form* (ABNF) ainsi que 10 fichiers de grammaire dans un format semblable à celui utilisé pour définir des grammaires CFG. Les deux ensembles de grammaires fournis reconnaissent le même ensemble de phrases.

Dans le cadre de ces directives, nous utiliserons une grammaire simple (B.1), qui n'est pas dans le corpus de phrases, afin de vous proposer quelques exemples de corrections. Cette grammaire permet de savoir s'il y a du soleil ou de la pluie à l'extérieur. La grammaire est au format ABNF, afin de réduire la grosseur du texte.

```
public $temperature =
    [$intro] ($soleil | $pluie) [$conclusion]
;

private $intro =
    il [fait | y a] | c'est
;

private $conclusion =
    à l'extérieur | dehors
;

private $soleil =
    [très] beau | soleil
;

private $pluie =
    pleut | [de la] pluie | pluvieux | nuageux
;
```

Figure B.1 Grammaire ABNF utilisée comme exemple pour les directives d'annotations

Nous présentons ici-bas une liste énumérant les différentes règles que vous devez suivre

lors de votre évaluation du corpus.

1. L'évaluateur doit utiliser seulement l'insertion, la mutation ou la suppression d'un mot et annoter chaque correction selon un format précis. Chaque opération correspond à une erreur même pour une mutation.

   (a) Insertion : On ajoute un plus (+) à la fin du mot à ajouter.
   il fait beau l'extérieur => il fait beau **à+** l'extérieur

   (b) Mutation : On met le mot corrigé en premier puis une barre verticale (|) suivi du mot original entre parenthèses.
   c'est pluveu dehors => c'est **pluvieux|(pluveu)** dehors

   (c) Suppresion : On ajoute un moins (-) à la fin du mot à supprimer.
   il fait vraiment très beau dehors => il fait **vraiment-** très beau dehors

2. L'évaluateur ne doit jamais corriger une séquence de mots qui serait reconnue par la grammaire. Si une séquence de mots est syntaxiquement valide par rapport à la grammaire, alors il ne doit pas y avoir de correction.
3. L'évaluateur ne doit jamais utiliser un mot ou une séquence de mots qui ne serait pas reconnu par la grammaire. Ceci veut donc dire que la correction doit pouvoir être analysée par un algorithme d'analyse syntaxique classique.
4. L'évaluateur doit corriger en essayant de comprendre ce que l'usager voulait dire en rapport avec la grammaire. Par exemple, si la phrase est « il fait vraiment dégueulasse dehors », même si le mot « dégueulasse » n'est pas dans la grammaire, l'évaluateur comprend qu'il ne fait pas beau dehors, il corrigera donc possiblement la phrase de cette manière : « il fait **vraiment- pluvieux+** dehors ». C'est donc dire qu'il utilise la compréhension sémantique pour faire la correction de phrases.
5. L'évaluateur doit essayer de corriger avec le moins d'erreurs possible. Donc, si l'évaluateur envisage deux corrections possibles, une avec deux erreurs et l'autre avec trois, il doit utiliser celle contenant que deux erreurs.

Les trois premières règles sont obligatoires, c'est-à-dire qu'elles ne peuvent pas être transgressées par l'évaluateur, car les algorithmes d'analyse syntaxique robuste utilisent ces trois règles lorsqu'ils font la correction des différentes phrases. Pour les deux autres règles, l'évaluateur utilise son jugement afin de déterminer quelle serait la meilleure correction à appliquer pour une phrase en particulier. L'évaluateur est l'étalon, il doit impérativement mettre la correction qu'il pense la meilleure en fonction de la grammaire.

Pour la règle #5, deux exceptions doivent être envisagées par l'évaluateur. Une première exception vient du choix entre une mutation ou une suppression suivi d'une insertion. Dans

ce cas, il utilise son jugement et les règles de la grammaire pour décider la meilleure correction possible. La deuxième provient du fait que quelquefois, l'utilisateur répétera deux fois sa réponse, mais la grammaire reconnait une forme telle qu'un seul ajout permettrait de reconnaitre la phrase. Dans ce cas, l'évaluateur utilise son jugement pour décider s'il doit utiliser la correction supprimant la séquence répétée ou ajouter un seul mot et donc utiliser la correction contenant le moins d'erreurs.

Il est conseillé de bien regarder les mots reconnus par la grammaire. Certaines grammaires sont plus compliquées que d'autres et certaines formes sont quelque peu non-orthodoxes. Par exemple, certains mots souvent coupés par l'utilisateur sont entrés tels quels dans la grammaire même s'il n'existe pas dans la langue française ou anglaise. Un exemple est le mot « immédiatement », qui se trouve dans une des grammaires, et qui est souvent coupé à « immé ». L'auteur de la grammaire a donc décidé de l'inclure dans la grammaire afin d'augmenter la reconnaissance des phrases ayant le mot « immé ». C'est pourquoi l'évaluateur doit vérifier adéquatement la grammaire lorsqu'il fait la correction des différentes phrases, car un mot qu'on penserait invalide pourrait être valide dans le contexte de la grammaire.

Afin de vous faciliter la vie, voici un petit protocole, complètement optionnel, donnant une façon de faire la correction des différentes phrases d'une manière à minimiser le temps.

1. Installer le plug-in *NuGram IDE Basic Edition* dans le logiciel *Eclipse*. Ce plug-in contient plusieurs outils pour la conception de grammaires ABNF. Il sera utilisé afin de faire la reconnaissance des phrases.
2. Ouvrir la grammaire ABNF correspondante dans le plug-in et démarrer l'interprétation en cliquant sur l'icône *Interpret Sentence*.
3. Ouvrir la vue *Test Data* qui permet d'entrer une phrase afin de voir l'arbre syntaxique, si la phrase peut être analysée par un algorithme qui ne fait pas de correction d'erreurs.
4. Entrer la phrase devant être corrigée dans la boîte de texte de la vue *Test Data*. Elle ne pourra pas être analysée dans sa forme actuelle puisqu'elle contient au moins une erreur.
5. Insérer, supprimer ou muter un ou plusieurs mots de la phrase afin de la rendre analysable en suivant les règles de correction.
6. Entrer la correction effectuée dans le fichier de benchmark, en respectant le format de correction requis.
7. Répéter le processus pour toutes les phrases puis pour chacune des dix grammaires du corpus de test.

ANNEXE C

RÉSULTATS BRUTS

Cette annexe contient les résultats bruts qui ont été obtenus durant la phase d'expérimentation. Ce sont les résultats qui ont été utilisés pour produire tous les graphiques et tableaux présentés dans la section résultats du mémoire.

Voici les résultats bruts pour le correcteur #1.

Tableau C.1 Algorithme de correction locale (Anderson-Backhouse) - Correcteur #1

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.7625	0.8878	1.67 ms	14.70 ms	0.34 ms
activation-date	0.7983	0.9139	19.81 ms	36.08 ms	0.37 ms
arrondissements	0.7685	0.9250	152.51 ms	791.35 ms	0.44 ms
attendant	0.7593	0.8484	20.17 ms	22.05 ms	0.24 ms
commands	0.6689	0.8560	10.87 ms	64.09 ms	0.30 ms
confirmation	0.8130	0.8640	1.94 ms	23.47 ms	0.34 ms
credit-card-expiration	0.5348	0.8445	16.17 ms	107.58 ms	0.54 ms
nas-dernier-triplet	0.5732	0.7474	617.29 ms	2396.98 ms	0.69 ms
postal-code	0.7955	0.9186	669.21 ms	1810.59 ms	2.55 ms
zip-code	0.8050	0.9358	4.19 ms	173.98 ms	0.96 ms
Moyenne	0.7279	0.8741	151.38 ms	544.09 ms	0.68 ms

Tableau C.2 Algorithme de correction globale (Lyon) - Correcteur #1

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.5590	0.7987	0.08 ms	28.02 ms	0.61 ms
activation-date	0.9077	0.9668	0.16 ms	226.12 ms	10.67 ms
arrondissements	0.8880	0.9690	1.27 ms	1395.15 ms	4.18 ms
attendant	0.7883	0.8259	0.24 ms	102.92 ms	1.32 ms
commands	0.7673	0.9145	0.33 ms	121.04 ms	1.68 ms
confirmation	0.7948	0.8755	0.10 ms	179.43 ms	2.26 ms
credit-card-expiration	0.7978	0.9290	0.34 ms	279.77 ms	9.76 ms
nas-dernier-triplet	0.6319	0.7876	1.09 ms	2410.50 ms	306.50 ms
postal-code	0.9311	0.9782	2.79 ms	16449.90 ms	48.20 ms
zip-code	0.7721	0.9300	0.13 ms	429.81 ms	4.87 ms
Moyenne	0.7838	0.8975	0.65 ms	2162.27 ms	39.00 ms

Tableau C.3 Algorithme de correction hybride (Hybride) - Correcteur #1

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.5676	0.7995	0.07 ms	27.02 ms	0.63 ms
activation-date	0.8597	0.9476	0.12 ms	201.91 ms	10.34 ms
arrondissements	0.8137	0.9495	1.15 ms	892.65 ms	10.49 ms
attendant	0.7755	0.8784	0.29 ms	86.42 ms	0.96 ms
commands	0.7540	0.9063	0.29 ms	88.99 ms	1.45 ms
confirmation	0.7877	0.8690	0.09 ms	135.22 ms	1.86 ms
credit-card-expiration	0.6569	0.8887	0.29 ms	152.02 ms	5.30 ms
nas-dernier-triplet	0.6322	0.7523	0.97 ms	1693.41 ms	127.78 ms
postal-code	0.9046	0.9650	2.56 ms	9243.91 ms	20.19 ms
zip-code	0.7988	0.9419	0.11 ms	293.55 ms	3.76 ms
Moyenne	0.7551	0.8898	0.59 ms	1281.51 ms	18.28 ms

Voici les résultats bruts pour le correcteur #2.

Tableau C.4 Algorithme de correction locale (Anderson-Backhouse) - Correcteur #2

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.8615	0.9316	1.34 ms	14.69 ms	0.35 ms
activation-date	0.8094	0.9254	20.48 ms	38.31 ms	0.38 ms
arrondissements	0.8085	0.9350	153.80 ms	796.79 ms	0.45 ms
attendant	0.7233	0.8711	20.08 ms	22.16 ms	0.24 ms
commands	0.6719	0.8327	10.87 ms	64.15 ms	0.31 ms
confirmation	0.8468	0.8884	1.91 ms	23.13 ms	0.34 ms
credit-card-expiration	0.6417	0.8675	15.87 ms	104.15 ms	0.54 ms
nas-dernier-triplet	0.5932	0.7647	610.96 ms	2413.14 ms	0.69 ms
postal-code	0.8158	0.9261	683.25 ms	1780.60 ms	2.57 ms
zip-code	0.8287	0.9646	3.99 ms	186.61 ms	0.98 ms
Moyenne	0.7601	0.8907	152.25 ms	544.37 ms	0.68 ms

Tableau C.5 Algorithme de correction globale (Lyon) - Correcteur #2

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.6467	0.8417	0.08 ms	28.41 ms	0.62 ms
activation-date	0.8899	0.9555	0.16 ms	224.76 ms	10.69 ms
arrondissements	0.9280	0.9810	1.26 ms	1399.45 ms	4.23 ms
attendant	0.7043	0.7863	0.23 ms	103.33 ms	1.33 ms
commands	0.7865	0.9112	0.34 ms	120.87 ms	1.69 ms
confirmation	0.8837	0.9379	0.10 ms	178.35 ms	2.25 ms
credit-card-expiration	0.8009	0.9399	0.34 ms	294.89 ms	9.83 ms
nas-dernier-triplet	0.6590	0.8081	1.10 ms	2433.13 ms	319.74 ms
postal-code	0.9733	0.9884	2.70 ms	16273.23 ms	49.14 ms
zip-code	0.8020	0.9600	0.13 ms	456.08 ms	5.10 ms
Moyenne	0.8074	0.9110	0.64 ms	2151.25 ms	40.46 ms

Tableau C.6 Algorithme de correction hybride (Hybride) - Correcteur #2

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.6667	0.8433	0.07 ms	26.93 ms	0.63 ms
activation-date	0.8419	0.9363	0.12 ms	203.02 ms	10.34 ms
arrondissements	0.8537	0.9615	1.15 ms	848.31 ms	10.61 ms
attendant	0.6915	0.8389	0.19 ms	84.72 ms	0.97 ms
commands	0.7338	0.8817	0.28 ms	85.76 ms	1.44 ms
confirmation	0.8748	0.9313	0.08 ms	131.02 ms	1.85 ms
credit-card-expiration	0.7261	0.9029	0.33 ms	149.00 ms	5.22 ms
nas-dernier-triplet	0.6687	0.7750	0.96 ms	1657.21 ms	134.82 ms
postal-code	0.9469	0.9752	2.54 ms	8732.01 ms	20.11 ms
zip-code	0.8287	0.9718	0.11 ms	290.74 ms	3.81 ms
Moyenne	0.7833	0.9018	0.58 ms	1220.87 ms	18.98 ms

Voici les résultats bruts pour le correcteur #3.

Tableau C.7 Algorithme de correction locale (Anderson-Backhouse) - Correcteur #3

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.8682	0.9350	1.40 ms	14.66 ms	0.34 ms
activation-date	0.8158	0.9310	20.45 ms	36.68 ms	0.37 ms
arrondissements	0.8405	0.9378	155.07 ms	797.75 ms	0.44 ms
attendant	0.6914	0.8474	20.77 ms	22.54 ms	0.24 ms
commands	0.6261	0.8219	10.99 ms	65.57 ms	0.30 ms
confirmation	0.8130	0.8706	1.99 ms	23.44 ms	0.34 ms
credit-card-expiration	0.4864	0.8330	15.96 ms	104.96 ms	0.54 ms
nas-dernier-triplet	0.5985	0.7609	652.01 ms	2409.01 ms	0.69 ms
postal-code	0.8236	0.9303	680.06 ms	1733.77 ms	2.54 ms
zip-code	0.8287	0.9646	4.08 ms	173.55 ms	0.97 ms
Moyenne	0.7392	0.8833	156.28 ms	538.19 ms	0.68 ms

Tableau C.8 Algorithme de correction globale (Lyon) - Correcteur #3

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.7467	0.8728	0.08 ms	28.96 ms	0.62 ms
activation-date	0.9325	0.9710	0.16 ms	229.86 ms	10.66 ms
arrondissements	0.9600	0.9838	1.28 ms	1462.91 ms	4.14 ms
attendant	0.7191	0.8139	0.23 ms	106.53 ms	1.48 ms
commands	0.8722	0.9674	0.33 ms	123.24 ms	1.67 ms
confirmation	0.9014	0.9543	0.10 ms	179.73 ms	2.28 ms
credit-card-expiration	0.7422	0.9198	0.34 ms	286.84 ms	9.65 ms
nas-dernier-triplet	0.6560	0.8065	1.08 ms	2544.36 ms	306.15 ms
postal-code	0.9471	0.9872	2.73 ms	17403.86 ms	48.46 ms
zip-code	0.8020	0.9600	0.13 ms	444.32 ms	4.85 ms
Moyenne	0.8279	0.9237	0.65 ms	2281.06 ms	38.99 ms

Tableau C.9 Algorithme de correction hybride (Hybride) - Correcteur #3

	Jaccard	Levensthein	Initialisation	Reconnaissance	Construction
activate-now	0.7267	0.8712	0.07 ms	26.88 ms	0.64 ms
activation-date	0.8845	0.9659	0.13 ms	200.36 ms	10.42 ms
arrondissements	0.8857	0.9644	1.16 ms	853.97 ms	11.22 ms
attendant	0.7064	0.8664	0.20 ms	85.21 ms	0.97 ms
commands	0.7967	0.9340	0.28 ms	85.91 ms	1.44 ms
confirmation	0.8943	0.9477	0.09 ms	132.71 ms	1.88 ms
credit-card-expiration	0.5709	0.8685	0.29 ms	148.52 ms	5.22 ms
nas-dernier-triplet	0.6657	0.7734	0.97 ms	1657.78 ms	130.71 ms
postal-code	0.9206	0.9740	2.55 ms	8787.27 ms	19.99 ms
zip-code	0.8287	0.9718	0.11 ms	284.96 ms	3.78 ms
Moyenne	0.7880	0.9137	0.58 ms	1226.36 ms	18.63 ms